Development of an Imager System Optimized for Low-Power, Limited-Bandwidth Space Applications

A Thesis

Presented to

The Faculty of

California Polytechnic State University, San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

By

Kalia Roseanne Glassey

April 2009

COMMITTEE MEMBERSHIP

TITLE: Development of an Imager System Optimized for Low-Power, Limited-Bandwidth Space Applications

AUTHOR: Kalia Roseanne Glassey

DATE SUBMITTED: April 8, 2009

COMMITTEE CHAIR: Dr. Fred DePiero, Assistant Dean, College of Engineering

COMMITTEE MEMBER: Dr. Jordi Puig-Suari, Professor of Aerospace Engineering

COMMITTEE MEMBER: Dr. James Harris, Director of Computer Engineering Program

ABSTRACT

Development of an Imager System Optimized for Low-Power, Limited-Bandwidth Space Applications

Kalia Glassey

A relatively new picosatellite standard, CubeSats have traditionally been used for simple educational missions. As CubeSats become more complex and utilize more complex sensors such as imagers, they gain enhanced credibility as satellite platforms.

Imaging systems on CubeSats have the potential to be used for a variety of uses, such as earth and weather monitoring, attitude determination, and remote sensing. However the size and power limitations of CubeSats pose an interesting challenge to the design of a capable, robust imaging system.

This thesis outlines the objectives and requirements of CP-3's imaging system, and describes the development process and methods. Test results from the imaging system are included, as well as lessons learned gleaned from CP-3's on-orbit operations.

This document can serve as a guideline for other teams wishing to develop imaging systems. While other developers may have different requirements or constraints, this roadmap illustrates each of the many considerations that must be taken into account when designing an imaging system.

ACKNOWLEDGMENTS

First and foremost, I would like to extend my appreciation to the entire PolySat and CubeSat teams, who taught me so much about real-world engineering. Dr. Jordi-Puig Suari has been an enthusiastic, supportive, dedicated project advisor. I owe special thanks to CP-3's software team, particularly Keith McCabe, David Cuddeback, and Kyle Leveque, who helped me in my struggle to learn how to program, and patiently answered questions.

To all the engineering faculty at Cal Poly who did an outstanding job teaching and inspiring, my thanks. In particular my thesis advisor Dr. Fred DePiero is a tremendous teacher who is knowledgeable, enthusiastic, and (above all, in this case) patient.

Finally, thanks to my family and friends who supported me at Cal Poly, and encouraged me to finish this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1: Introduction

1.1 CubeSat Project

The CubeSat project was started by Cal Poly San Luis Obispo and Stanford University to allow universities to have easier and cheaper access to space. Since its beginning in 1999, more than 60 universities and companies around the world have participated. By providing a picosatellite standard and coordinating launch opportunities, Cal Poly ensures that students and companies wishing to build small payloads have a convenient and cost-effective way of doing so.

There are several services that Cal Poly provides to the CubeSat community. The first is the use of a standard CubeSat specification so that developers have guidelines for the construction of their satellites. This standard details the physical size and mass requirements (10 cm cubed, and less than 1 kg), as well as ensuring that the satellite does not cause damage to other CubeSats or the primary payload [5]. Thus, each satellite must be designed to ensure that it can survive launch and space conditions. In addition, the standard calls for spring-plunger switches in the rails that ensure the satellite does not turn on until deployment.

To provide a standard interface to the launch vehicle, Cal Poly has also developed the Poly Picosatellite Orbital Deployer or P-POD. Each of these deployers can contain three standard CubeSats, one standard and one double-length, or one triple-length. The P-POD is flight-proven, and provides an easy way for launch providers to integrate small satellites as secondary payloads.

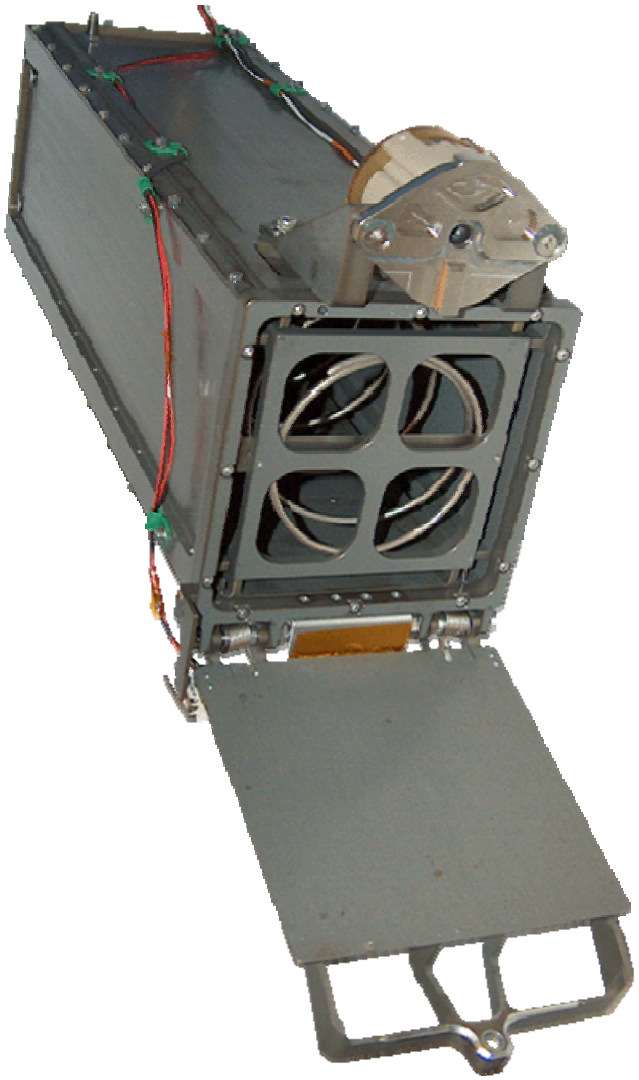


Figure 1-1: Poly Picosatellite Orbital Deployer (P-POD)

In the US market, a company or entity wanting to launch a payload must typically purchase the entire launch vehicle. That entity then has little incentive to allow secondary payloads on board, seeing them as added risk because of their potential to harm the primary payload. This makes domestic launch opportunities expensive and rare. Because foreign launch providers usually retain ownership of the launch vehicles, they have proven more willing to work with small university satellites in order to defray

launch costs by putting more payloads on board. Thus most CubeSat launches have taken place from Baikonur Cosmodrome in Kazakhstan. Since satellites are considered weapons under US law, Cal Poly has had to seek status as a registered arms dealer to be able to obtain the necessary export licenses. Without this service, any US-built satellites would find it difficult to obtain the necessary permissions to utilize foreign launch opportunities.

To ensure that the satellites will survive the launch conditions, as well as space conditions, Cal Poly provides acceptance testing and integration services. The satellites are tested on a vibration table to 150% of NASA standards, which exceeds the amount of vibration expected during launch. In addition, Cal Poly's thermal/vacuum chamber allows for testing of space conditions. Satellites are placed in the vacuum chamber, and the temperature is cycled to simulate the hot and cold conditions seen during successive orbits. Finally, Cal Poly integrates the satellites into the P-PODs at Cal Poly before shipping them to the launch site and supervising integration with the launch vehicle. In addition, Cal Poly provides telemetry to the satellite developers after a successful launch so that they can track and communicate with their satellites.

As is apparent, many of these steps would be beyond the capabilities of most university satellite developers. Even companies that do have greater resources have shown their desire to take advantage of this program for small experimental payloads as a way of testing prototypes.

Although each university has individual motives for wanting to develop a CubeSat, the CubeSat team at Cal Poly has found that there are a few main reasons for participating in this program. For some universities, such as Cal Poly's own PolySat

program, the desire is to gain experience in systems-level design and building an actual working satellite. For these types of developers, the actual payload content is not as important as the experience gained through the design and fabrication process. For other universities, the primary mission is a scientific experiment, and the satellite is just a means to an end.

1.1.1 History of Project

1.1.1.1 Project Foundation

Since the inception of the CubeSat project in 1999, there have been a total of 38 CubeSats launched, 24 of which made it to orbit. The other 14 were on the July 2006 failed Dnepr launch. Cal Poly has been heavily involved in three launches, two Dnepr cluster launches in July 2006 and April 2007 and a Minotaur launch of NASA Ames' GeneSat-1 in December 2006 [19].

1.1.1.2 Cal Poly Satellites

In 2000, Cal Poly began to develop its first CubeSat, CP-1. The design was purposely kept as simple as possible and the goal was to have a very reliable system that would give the project experience in building satellites, as well as help us better understand the conditions faced in orbit.

The second satellite, CP-2 was much more ambitious. The goal was to develop a standardized bus that could accommodate many types of payloads. Thus the design was much more sophisticated, although the developers still strived for simplicity and redundancy where possible.

CP-3, begun in 2005, built on the standardized bus built for CP-2. Many more sensors and devices were incorporated into the design to allow us to collect as much information as possible about the space environment, as well as to learn how to accurately determine (and control) position and orientation.

1.1.1.3 Other Developers

The first developers were virtually all universities; government and corporations did not see much potential in such a small satellite. Some of the pathfinders were the University of Toronto, the Technical University of Denmark, the University of Aalborg (Denmark), Stanford University, the Tokyo Institute of Technology, and the University of Tokyo. These organizations were represented on the first launch of CubeSats in June 2003 [19].

As more and more missions have flown, other developers have started to see potential in this new form factor. Boeing developed a CubeSat, CSTB-1, as did Aerospace Corp., Tethers Unlimited, and QuakeFinder. Recently NASA Ames' 3U GeneSat was a huge success, and has encouraged NASA to build more CubeSats.

1.1.1.4 Recent Launch History and Results

Although there were few launches for many years, over the last 3 years, the pace of launches has increased significantly. August 2006 saw Cal Poly's first Dnepr launch with 14 satellites, which unfortunately ended up in the Kazakh desert. December 2006 saw the first US CubeSat launch, GeneSat-1 on a Minotaur launch vehicle. In April 2007 Cal Poly coordinated another Dnepr cluster launch from Baikonur Cosmodrome, which this time was successful. Although previous launches had seen perhaps a 50% success

rate, every single satellite has communicated (at least briefly) in those two launches, further validating the CubeSats' viability. Most recently the Indian Space Research Organization launched 6 CubeSats into space in April 2008.

1.1.2 Current Development and Research

As CubeSats gain in popularity among corporations and government as well as universities, there is an increased focus on increasing capability and moving beyond CubeSats as educational projects. A few of the most challenging problems impeding CubeSats from becoming more widely used are described below. Many developers are working to come up with better solutions to these problems in order to increase the utility of CubeSats.

1.1.2.1 Communication Systems

Lack of high data rate communications is a serious impediment to widespread use of CubeSats. Although many people have become convinced that useful missions can be accomplished within the space available, the inability to rapidly download data has hindered development efforts. This problem can be traced to three sources: regulatory, power, and available technology.

Because of the difficulty in obtaining FCC licenses (much time and money is needed), university CubeSats have traditionally gone to AMSAT (the Radio Amateur Satellite Corporation) to obtain global licenses in the amateur bands. Problems with this approach include resistance from ham radio operators who resent the use of their band by satellites with questionable utility to the amateur radio community, the relatively low frequencies available (~430 MHz), and the restrictions on commercial and government

satellites. The FCC offers experimental licenses with expedited application processes, however these still have significant drawbacks. It took Cal Poly over 2 years to receive an experimental license for CP-1. When a license was finally received, it was restricted to US airspace (a problem for satellites that do not know their position to any sort of accuracy), had a very limited time of use, and was tied to a particular launch and orbit. In addition the FCC is starting to more stringently enforce their 25 year lifetime requirement for satellites, which is a problem for those that cannot control their altitude. To get around this, some recent satellites (GeneSat-1, MAST) have used 2.4 GHz frequencies which are in the unlicensed ISM band. The problem with this frequency is that there is little off-the-shelf equipment suitable for space. In the case of these satellites, significant handshaking time limited the amount of data retrievable. Ultimately the regulatory issue will have to be solved, but it's beyond the scope of any one university's program, so it remains an issue.

Besides regulatory issues, power is a significant constraint. Because a 1U CubeSat only generates an average of 1-2 W, with a peak power output of about 5 W, most of the link budget needs to be made up on the ground. While Cal Poly and others have been able to use SRI International's 60-foot dish to great advantage, it is difficult for most schools to have access to this kind of equipment.

Because RF and communications are one of the most difficult parts of building a satellite, developers have tended to rely on off-the-shelf systems. Unfortunately, most are not designed for satellite applications, and so the selection is quite limited in frequency and capability. Developing CubeSat-compatible radios is currently a hot topic of research. Software defined radio is especially interesting because of its flexibility.

1.1.2.2 Attitude Determination and Control Systems

Many of the new satellites under development consist of ADC experiments. With each round of satellite development, the developers' knowledge, experience, and confidence have increased. While the first satellites, such as Cal Poly's CP-1, were primarily concerned with very basic functions such as C&DH and communication, satellite missions have steadily become more sophisticated.

At this point there is a general consensus that before more interesting missions can be attempted, a reliable ADC system must be developed. While this is a mostly solved problem for larger satellites, the small size and power constraints make it quite a challenge on CubeSats.

Although standard methods of attitude determination using gyros, sun sensors, magnetometers, etc. exist, currently one of the most pursued areas of development is in using an imager to do attitude determination. This can be done in two ways, either through horizon sensing or through star tracking. In horizon sensing, the amount of Earth that can be seen in images is used in conjunction with an orbit propagator to determine which way the satellite is pointing. Star tracking is somewhat more versatile. It compares star images taken by the satellite to a star catalog and uses an algorithm to determine which way the satellite is facing.

1.1.2.3 Imaging Systems

Imaging systems are also of great interest to developers. Although they are not going to compete with large reconnaissance satellites anytime soon, there is a great deal of interest, both simply from a desire to see an image from "your" satellite, and from

other potential missions. Most developers thus far have gone with a "camera on a chip" option. These systems, typically developed for cell phone use, are quick and easy to integrate into a project, but limit flexibility.

1.1.3 Motivation of Thesis

As noted above, there is increasing interest in star tracking, as well as space imaging in general. However, CubeSats suffer from several constraints not necessarily present in larger satellites. One of the foremost of these is limited power and bandwidth, which makes the downlink of large image files problematic. Given the paucity of CubeSat launches, developers typically must accept whatever orbit is offered, most of which are sun-synchronous, low earth orbits where downlink times can be as short as 10 minutes a pass or roughly 40 minutes a day. Thus image compression is necessary to make it possible to download images in a reasonable amount of time.

This thesis is intended to demonstrate the development and test of an imaging system for a CubeSat platform. As such, it will explore the design philosophy and practices as well as testing methods. A major component of this thesis will be to explore the optimal way of compressing space images losslessly. Star images were chosen as test cases, because they are the most likely to be taken in space, the most useful for star tracking, and because they are possible to obtain with identical hardware on the ground.

Although lossless compression algorithms are relatively well understood, the specialized nature of the images under consideration makes it a reasonable hope that a combination of techniques can provide a compression schemed optimized for this application.

1.2 Thesis Goals

The goal of this thesis is to develop and test an imaging system for a CubeSat. The imaging system design will include imager selection, hardware design and layout, software design, image compression, and testing. Of these, this thesis will focus on imager selection, hardware design and layout, and image compression, as these are the areas in which the author had lead responsibility and performed the substantial majority of the work. The remaining components, software design and testing, will be treated in the appendices. The end result will be an imaging system for CP-3 to be launched at the next available opportunity. While CP-3's constraints and requirements may not be the same as those of other developers, it is hoped that this thesis can serve as a guideline for other teams wishing to develop CubeSat imaging systems.

1.3 Thesis Organization

This thesis is organized into eight chapters, with appendices containing hardware architecture and specifications, payload software information, program code, test images, and test results.

In the first chapter, the CubeSat project is introduced and the motivation for the thesis presented.

The second chapter contains an imaging system overview and discusses the development philosophy and the systems level considerations that went into the imaging and compression system design.

Chapter 3 discusses the details of the imaging system development, including hardware selection, system design and layout, and software development.

Chapter 4 describes the integration and testing procedures undergone by the imaging system, including functional testing as well as vibration and thermal/vacuum testing.

Chapter 5 contains an overview of information theory, which provides much of the theoretical background to the field. Quantitative and qualitative metrics for evaluating different compression techniques are explained. A survey of different image compression techniques is also presented, along with their advantages and disadvantages.

In Chapter 6 the compression system development is described. There is explanation for why the algorithms considered were selected, as well as the reasoning behind ruling others out. Details of implemented algorithms are described.

Chapter 7 is the conclusion and summarizes the thesis process and results. The results of the various compression tests are summarized and the optimal scheme is presented.

Chapter 8 is a short epilogue that discusses the fates of the Cal Poly satellite, projects currently under development, and lessons learned.

Appendices A through H provide additional background and technical information and results. In Appendix A key hardware specifications are presented; Appendix B shows CP-3's bus and payload architecture; Appendix C details the payload software development; Appendix D presents imaging system testing data; Appendix E contains the compression test program source code; Appendix F shows the sample test images; Appendix G gives the results of the compression test program.

Chapter 2: Imaging System Overview and Systems Level Considerations

This section describes the basic components of an imaging system and discusses the development philosophy and systems level considerations that went into the imaging system design.

2.1 System Diagram

Figure 2-1 below shows a generic satellite imaging system. The payload processor controls imager settings and handles the transfer of the completed image to non-volatile memory, ideally through a DMA (Direct Memory Access) transfer. Non-volatile memory is important to ensure that loss of power does not delete image data. The payload processor also has control/data flow with the bus and/or communications processor. The communications processor receives the image data from the non-volatile memory and encodes it for downlink to the ground station. This encoding may include any or all of the following: source coding (compression), channel coding (error-checking code generation), and data packetizing. The image data is downlinked to the ground station during a satellite pass and the data is decoded.

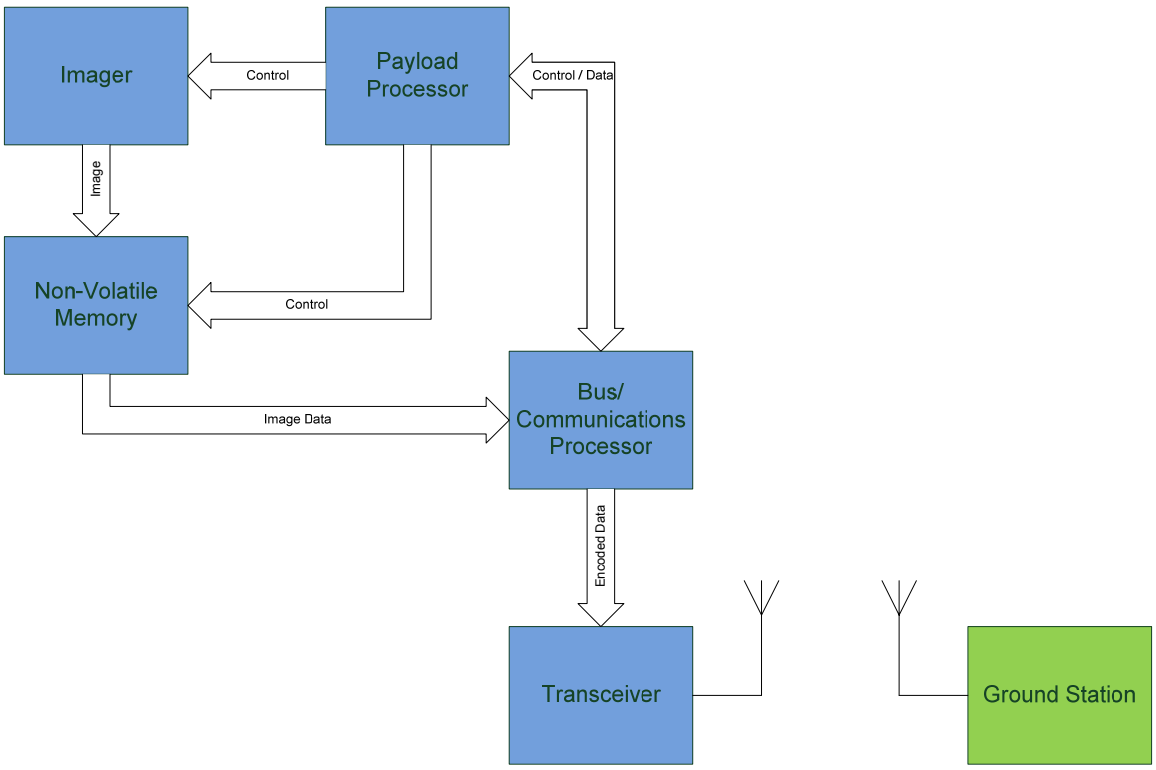


Figure 2-1: Generic Satellite Imaging System

2.2 Previous Work

Previous imaging systems by Denmark's Aalborg University [1] and others had used "cameras on a chip" for imaging. These are small, imaging units with included hardware and firmware such that it is very easy to integrate into a system and begin programming work. Such units tend to be low resolution (generally VGA, 640x480 pixels) and have fewer adjustable imaging parameters. For example, they may not allow the user to adjust the pixel array integration time. In addition, the overall size of the chip tends to be larger than necessary, and its shape and form factor may not be optimal for an individual satellite's use. Moreover, many companies do not make their CMOS imagers available in this prepackaged format, limiting the selection.

2.3 Development Philosophy

Because of the limitations of "cameras on a chip" noted above, the author opted to develop an imaging system from scratch, designing custom hardware. The main motivation for doing this was to obtain a more versatile, flexible system and to enhance the learning process.

Designing a custom imaging system enabled more versatility in use. While camera on a chip systems are relatively easy to get working, they do not allow full control of all parameters, and are not available for all imaging sensors. In particular, it was necessary to have an imaging sensor with high resolution and light sensitivity, and restricting the selection to cameras on a chip would have constrained that choice.

Although the reasons for downloading images vary with the mission, typically lossless compression is desired. For example, in the case of a star tracker, it is desirable to have identical images on the ground to those used to process the star tracking algorithm in the satellite. In this way it is assured that the algorithm is operating as intended. In addition, because this imaging technology is somewhat unproven in space, it is necessary to have the identical images to obtain a better sense of their advantages and limitations. For these reasons only lossless compression was considered for the design.

Other constraints on a compression system include the more limited processing capabilities of typical microprocessors. Thus, in addition to compression ratio, the feasibility of implementing the compression code in a microprocessor was taken into account. Compression algorithms that were simple and easy to implement were preferred.

2.4 Systems Level Considerations

Several systems level considerations informed the sensor selection and board design process.

2.4.1 Power, Size, and Mass

First, the imaging system needed to fit within the power, size, and mass constraints of the satellite system. Realistically speaking, the imager would only have to be on for several minutes out of each day, so average power was less of a concern than peak power. Size and mass for most imagers were comparable, but were definitely a significant concern when choosing a lens that offered both a small field of view and was not excessively large or heavy.

2.4.2 Integration Considerations

Integration considerations included both software, hardware, and structural components. In order to keep the design as simple as possible, the imaging system needed to be designed with minimal software complexity. In practice this meant that systems with automatic data transfer such as DMA were preferable, as were bus protocols that were already in use on the satellite bus such as I2C. Structurally the system needed to fit within the available payload envelope and offer visual pathways for the fields of view of the imagers. Figure 2-2 below shows four candidate imager placement concept sketches (pink cylinders shows imager field-of-view), of which the fourth was ultimately selected for its simplicity and avoidance of complicated brackets.

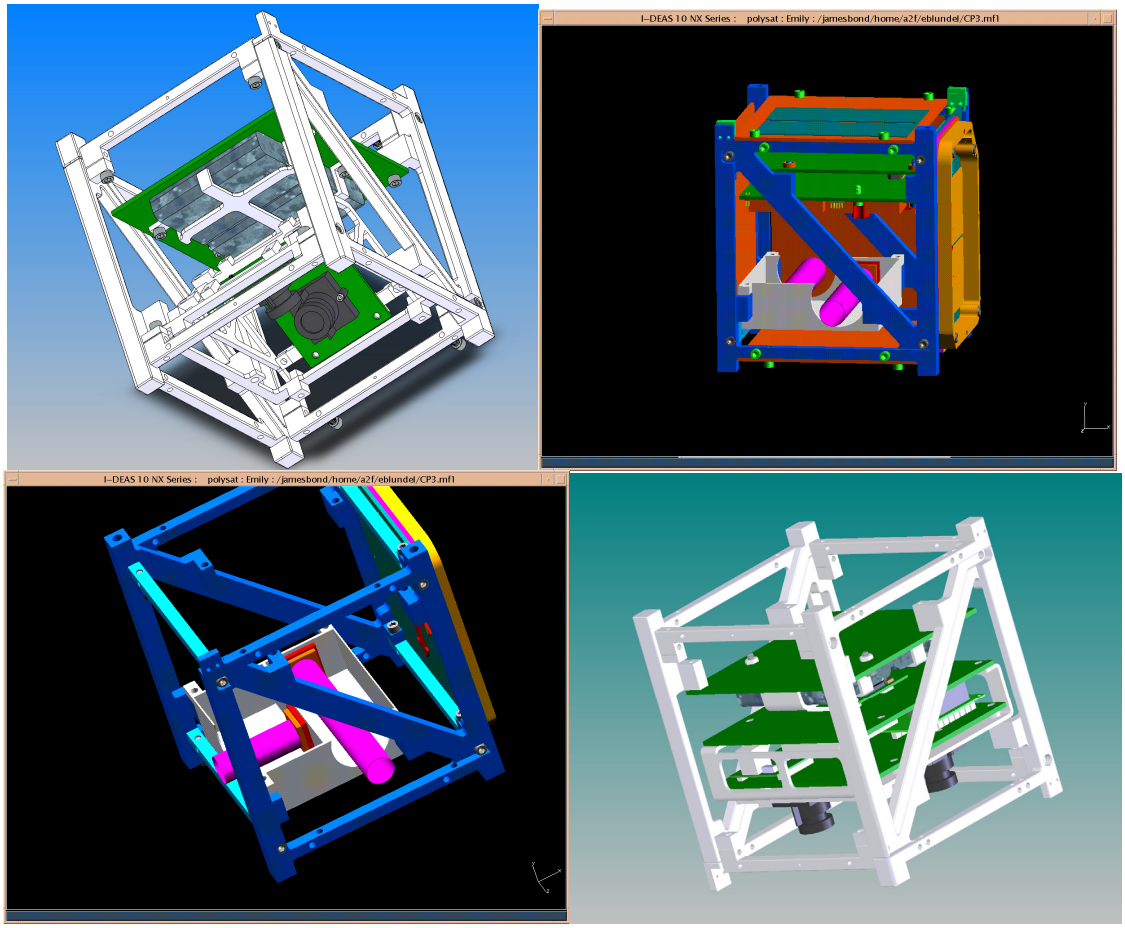


Figure 2-2: Candidate Imager Placement Concept Sketches

2.4.3 Imager and Lens Specifications

One of the paramount considerations when choosing imagers was their sensitivity. Because the team wanted the ability to take highly detailed star images, imagers that could use more of the incident light were highly preferred. In addition to this, it was desirable to use an imager that had many adjustable parameters such as gain and exposure time. Since it wasn't possible to accurately test what the imager would see in space on the ground, it was necessary to be able to adjust these parameters and calibrate the system on orbit.

Lens characteristics were also very important. In addition to the size and mass considerations mentioned above, lenses without distorting effects (such as fisheye) were necessary. Also, in order to optimize the amount of light hitting the imaging array, a lens with a relatively small field of view that would focus the light was necessary. Another consideration was the desire to avoid plastics and lens films, due to their unknown reactions to a vacuum environment and temperature extremes.

2.4.4 Compression Considerations

Despite the existence of many good compression algorithms in existence, the unique requirements for CP-3's image compression algorithm dictated a custom approach. The requirements were as follows:

- lossless compression—to allow for the best experimental results, a pixel-for-pixel accurate image was necessary
- computing requirements—because of limited computational ability and memory, the algorithm needed to be simple and easy to implement
- fixed-point compatibility—in order to simplify software development, algorithms requiring floating point calculations were not considered
- algorithm simplicity and flexibility—in order to fit within memory constraints while still allowing for algorithm tailoring to the space images' unique characteristics
- packetizing compatibility—ideally the algorithm would be robust enough to allow some image decoding even if not all packets were received

The unique characteristics of a CubeSat imaging system provide several limitations not found in ordinary compression systems. However, there are certain advantages, not least of which is the predictable nature of image contents. Following are several constraints and some advantages that played a significant role in the compression approach.

2.4.4.1 Power

The payload processor takes up a substantial part of the satellite's power budget, meaning that time spent compressing data both keeps the payload from performing other experiments, and also uses valuable power. So the compression algorithm should take as little time and processing power to compute as possible. Weighed against this, however, is the power needed to communicate the image back to earth, which depends directly on the compression ratio.

2.4.4.2 Bandwidth

This is possibly the most significant constraint. At CP-3's downlink rate of 1200 baud, an uncompressed 8-bit image of 1280 x 1024 pixels takes 145 minutes to download. Clearly the compression ratio needs to be as high as possible to allow the fastest possible image download.

2.4.4.3 Processing Capabilities

Most small microprocessors suitable for CubeSat usage have somewhat limited processing capabilities. Thus any compression scheme must take into account limited speed, memory, and precision. For example, the CP-3 processor does not have hardware

implementation of floating point computation. Thus any floating point computations run more slowly than fixed point.

2.4.4.4 Downlink Time

With CubeSat orbits dependent on primary payload requirements, daily downlink times can be quite limited. For equator orbits, earth station locations closer to the poles may get a very limited amount of downlink time. For polar orbits, although each point on earth is covered, there are only two sets of passes per day. At low earth orbits, each orbit is approximately 90 minutes, with two passes typically available in each set. Thus there are only about 40 minutes of downlink time available per day. For the example in section 2.4.4.2, one image would take at least 3.5 days to download.

2.4.4.5 A Priori Knowledge

Unlike many systems in which it is impossible or impractical to calculate probabilities for certain compression methods, it is easy to do it in this case by transmitting this data along with the compressed image. This enables the use of Huffman coding, as well as more efficient LZW coding.

2.4.4.6 Non-Real Time

Because the system is not real-time, many processing-heavy compression systems are possible. If there were time constraints, or video, some of the schemes implemented would not be possible.

Chapter 3: Imaging System Development

Because of the relatively low power available, and the large amounts of data generated by imagers, the development of an imaging system for a CubeSat is much more than simply the choice of an imager. Both peak and latent power need to be considered, as well as how to compress and package the data for the most efficient download.

3.1 Hardware Selection

In selecting the hardware for the imaging system, the imager is naturally of paramount consideration. Also important and often overlooked, however, is the interface to a processor that can rapidly process the signal.

The first question was whether to buy a camera-on-a-chip system to aid in limiting software complexity. Although it was a tempting option, it did not afford enough flexibility in the system. In addition, camera-on-a-chip systems are typically more geared toward video, and did not have as many options for controlling levels, exposure, etc., which would limit the ability to take high-quality still images.

3.1.1 Imager

Most CubeSat developers have moved away from the traditional CCD (charge coupled device) imagers, and primarily choose imagers using CMOS technology. Traditionally, CCD imagers have been used for astronomical observation because of their much higher quantum efficiencies—the amount of light that is converted into electronic signal by the array. However, they have a significant drawback for the CubeSat application in that they are much more power hungry, and take much longer to process an

image than a CMOS sensor. In addition, because of the proliferation of cell phone cameras, there is a large variety of low-cost CMOS imagers available. For this reason, the choice of imager was restricted to one that was a CMOS sensor.

Several candidates were selected, and a worksheet tabulating their characteristics was made for comparison (see Figure 3-1 below). Most of the devices used a two-wire bus for control lines, and either an 8- or 10-wire parallel data bus. The most important criteria were sensitivity, resolution, interface, and power consumption. Ultimately the choice was made to go with the 1 Megapixel grayscale Kodak KAC-9638. In addition, the color version, KAC-9648 was included on the final design for redundancy and comparison purpose. Both of these parts used an I2C interface, which made integration into the existing I2C bus system much easier. See Appendix A for a summary of the KAC-9638 and KAC-9648 datasheet.

Part	OV3610	OV2610	MT9T001	IBIS4-6600	STAR-1000	VC5700	KAC-9638/ 9648	ICM200E
Color Version?	Yes	Yes	Yes	Both	No	Both	Both	Yes
Manu.	Omni-Vision	Omni-Vision	Micron	FillFactory	FillFactory	STM	Kodak	IC Media
Chip Size (mm^3)	14.5 x 14.5 x 2.5	14.22 x 14.22 x 2.23	14.22 x 14.22 x 2.25	24.13 x 24.13 x 3.048	?	?	14.22 x 14.22 x 2.58	?
Package Type	CLCC-48	CLCC-48	PLCC-48	CLCC-68	84 pin J-leaded	CLCC-48	LCC-48	CLCC-48
Array Size (pixels)	2048 x 1536	1600 x 1200	2048 x 1536	3002x2210	1024 x 1024	1600 x 1200	1032 x 1288	1600 x 1200
Power Supply (V)	3.3	2.5 & 3.3	3.3	2.5	5	3.3	3	2.5 & 2.8
Active Power (mW)	132	165	792	200	400	165	180	130
Standby Power (µW)	33	33	825	not specified	not specified	495	1500	60
Pixel Size (μm^2)	3.18	4.2	3.2	3.5	15	4	6	3.45
Image Transfer Rate (f/s)	7.5 (min) 78 (max)	10 (min) 40 (max)	12 (min) 93 (max)	5 (min) ? (max)	12 (min) ? (max)	20 (min) 40 (max)	18 (min) ? max	20 (min) ? max
Dynamic Range (dB)	60	60	61	61	72	not specified	55	59
Output	10-bit digital raw RGB	10-bit digital raw RGB	10-bit digital raw RGB	10-bit digital raw RGB	10-bit digital raw RGB	10-bit digital raw RGB	8 or 10-bit digital raw RGB	10-bit digital raw RGB
Lens Size (in)	0.5	0.5	0.5	0.5	not specified	0.5	0.5	0.385
Sens-itivity (V/Lux*s)	0.9	1	1	1.57	not specified	not specified	2.5	1.4
SNR (dB)	40	54	43	not specified	not specified	41	not specified	45
Image Area (mm^2)	6.51 x 4.88	6.72 x 5.04	6.5 x 4.92	10.51 x 7.74	15.4 x 15.4	6.48 x 4.864	6.192 x	5.52 x 4.14
Oper. Temp. (°C)	0 to 40	0 to 40	0 to 60	0 to 50	0 to 60	0 to 40	-10 to 50	not specified
Storage Temp. (°C)	-40 to 125	-40 to 125	not specified	not specified	-10 to 60 (not longer than 1 hour)	not specified	-40 to 125	not specified
Software Protocol	SCCB (Serial Camera Control Bus)	SCCB (Serial Camera Control Bus)	2 Wire Serial Bus	3 Wire SPI (Series to Parallel Interface)	3 Wire SPI (Series to Parallel Interface)	I2C	I2C	2 Wire SIF

Figure 3-1: CMOS Imager Comparison Chart

3.1.2 Payload Processor

Although the author of this thesis was not primarily responsible for the selection of the payload processor, she interacted with the software team in order to ensure that it met the imaging system requirements. The much higher computing power needed to process signals from the imager dictated a much more capable payload processor than anything the team had used before. Previously the Cal Poly satellites had used PIC microprocessors for Command and Data Handling (C&DH), communications, and the payload. In this case, however, it was felt that an actual digital signal processor would be better for fast and efficient processing of the signal. Figure 3-2 below shows the processor comparison chart used to make the decision.

	PIC 18	dsPIC 30	Gumstix	Blackfin
High Level Languages	C	C	C, C++	C, C++, EC++
Assembly Languages	PIC 18 Assembly	dsPIC 30 Assembly	ARM	Blackfin Assembly
IDE	MPLab, IAR	MPLab, IAR	gcc + gdb + vi	VisualDSP++, MULTI® 2000 (Green Hills)
Clock Speed (MHz)	up to 40	20 or 30	200 or 400	200, 300, or 350
Data Memory (kB)	3	8	34 + 6400 (external)	32 L1 + 4 L1 "scratch" + 256 L2 + TBD external
Code Memory (kB)	128	144	4000 flash + MMC card	TBD by external devices
EEPROM (kB)	1	4	none	none
Architecture	8-bit ALU, 8-bit multiplier	40-bit ALU, 17-bit multiplier, 40-bit shift, 2 40-bit accumulators	?-bit ALU, 16-bit multiplier, 40-bit accumulator	2 40-bit ALUs, 2 16-bit multiplier/accumulators, 40-bit shift, 4 8-bit video ALUs, 2 40-bit accumulators
Compiler	mcc18, IAR, CCS	?	GNU gcc	Bundled w/ VisualDSP
Programmer	ICD	ICD	Serial Bootloader	JTAG
Emulator	ICE 2000/4000	ICE 4000	JTAG	JTAG
Operating System	none	none	Linux	Linux
Operating Voltage (V)	2 to 5.5	2.5 to 5.5	3.6 to 5.2	1.5 core, 3.3 external
Peak Power (mW)	483	?	825	?
Average Power (mW)	180.6	240	297 to 495 depending on clock speed	?
Operating Temp (°C)	-40 to 125	?	0 to 85	-40 to 85
Size (mm)	12 x 12 x 1	31 x 31	20 x 80 x 6.5	31 x 36 x 6.1

Figure 3-2: Processor Comparison Chart

The candidates were quickly narrowed down to the Gumstix, a tiny fully-functional computer the size of a pack of gum, and Analog Devices' Blackfin DSP. The Gumstix running embedded Linux could make programming a much easier task, but it also drew much more power than the Blackfin.

Before the final decision was made, thermal/vacuum and vibration testing was undertaken to see if one (or more) of the candidates would not survive the launch or space environment. In particular, the connectors for both appeared rather flimsy, and vibration was a serious concern. Both processors passed the thermal/vacuum test without problems, but the Blackfin encountered problems during the vibration test. The Gumstix with the aid of a makeshift bracket managed to survive the vibration testing, as did the Blackfin when a bracket holding it in place was added (see Figure 3-3 below).

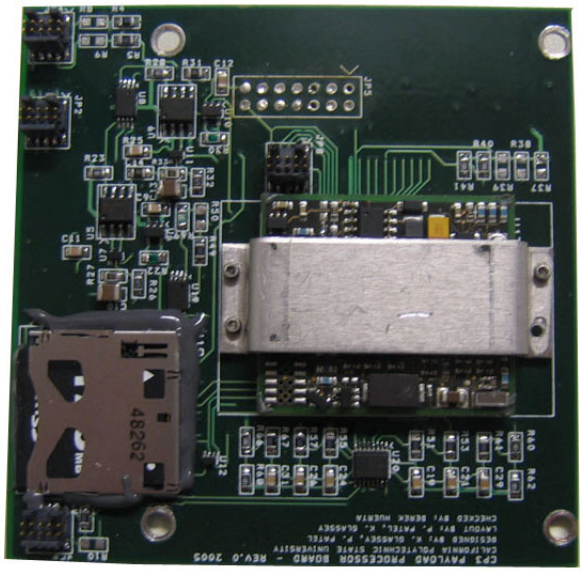


Figure 3-3: Payload Board Showing Processor Bracket (center right)

Ultimately the extra complexity and power draw of the Gumstix did not seem justifiable given the relative modesty of the requirements for the payload processor. Although the Blackfin looked more promising, it was only possible to buy the processor alone from Analog Devices. This would entail a great deal of design and layout time to interface the memory, and other peripherals. For that reason, the discovery of the Austrian company BlueTechnix' completely integrated, ready-to-use Blackfin-based DSP chips was fortuitous. A variety of hardware peripherals was available on the BlueTechnix Tinyboards, including flash memory and a DMA system, enabling automatic readout from the imager.

The development of the payload and imaging system software (with the exception of the image compression software written for this thesis) is discussed in Appendix C.

3.1.3 Lenses

Choosing an appropriate lens was also an important aspect of hardware selection. The main considerations were to select a lens that had a relatively small field-of-view, did not distort the image (no fisheye effect), let as much light through as possible, and could be focused to the near-infinite field required for crisp images of the earth and stars. Also critical was ensuring that the lens holder, lens, and lens films did not outgas unacceptably during thermal/vacuum conditions. Several lenses were considered and tested before the final lens was selected.

3.2 System Design and Layout

Because of the space constraints, the system design and layout was much more critical than it might be in another application. In particular, the total payload volume,

including imagers, processor, and lenses, could take no more than about 1/3 of the total 10 cm^3 volume. The final integrated satellite, including payload, but with one side panel removed is shown in Figure 3-4 below.

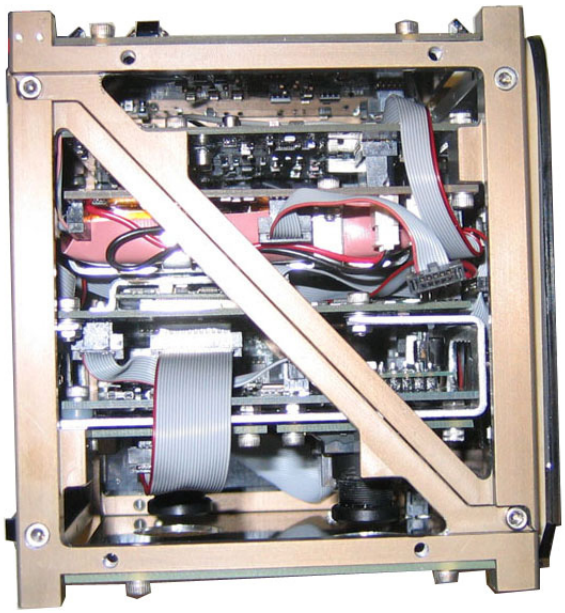


Figure 3-4: CP-3 Interior (camera lenses near bottom)

3.2.1 Imager Considerations

Although the imager itself is a single tiny chip (about 1 cm^2), supporting electronics were necessary for correct functioning. The first question that presented itself was whether or not to use the headboard available from Kodak directly in the satellite, or to design a new layout. The benefit to the protoboard was that it plugged directly into a Kodak development board that interfaced with a computer, making testing and debugging easy. However the headboard itself was rather bigger and bulkier than it needed to be, and significant space savings could be achieved by creating a custom layout.

In order to gain the benefits of both the Kodak development board and a smaller size board, the author decided to make a custom daughterboard that could both plug directly into the Kodak development board as well as be installed directly on the payload board (see schematic in Appendix B). However, though the design was based directly on Kodak's own headboard, the daughterboard did not correctly interface to the Kodak development board. The suspicion is that differing trace lengths may have led to timing errors on the high-speed parallel port making the daughterboard unusable in this fashion. Nonetheless, once the daughterboard was fully integrated into the payload, it was possible to adjust settings and take pictures.

As with many of the payload and bus sensors, a temperature sensor was placed near the imager to ensure that the imager temperature was known when a picture was taken. As is described in Chapter 4 below, thermal/vacuum testing showed that the average black level of the imager was significantly higher at elevated temperatures. Having a temperature sensor meant that it was possible to calibrate out the extra value.

Both the KAC-9638 and the KAC-9648 had the capability of putting the data output pins into tri-state mode. This enabled the connection of both imagers to the same DMA pins. To use one imager, it was necessary simply to toggle the other imager's output pins to tri-state values.

3.2.2 Payload/Satellite Bus Interface

Besides the physical interface to the satellite, which was dictated by structure dimensions, the electronic interface needed to be determined. Because of the relatively large power draw of the payload (largely due to the processor's power requirements), the

processor needed to be capable of being turned off, which dictated some sort of master/slave configuration. The Blackfin had a built-in UART module, but the PIC processor used for C&DH did not, meaning that it would have to be written from scratch if this was desired.

In the end, a compromise solution was worked out: power to the payload could be toggled as desired by the C&DH processor, the satellite bus's pre-existing I2C was used between the C&DH and the payload processors, and UART was used to talk directly to the processor from the development bench. The hard-off option used to control power to the payload was certainly the simplest, but ended up causing problems down the road. In particular, the team found that having the processor turned off but still connected to the I2C bus network caused communications problems on the I2C bus. This problem ultimately had to be solved by adding isolation buffers on the I2C lines to the payload. The UART connection between the development bench and the board turned out to be very useful and led to much easier debugging and programming of the payload without having to go through an I2C interface.

Chapter 4: Imaging System Integration and Test

Because the PolySat team had little experience with any of the major components being considered for the payload, they were tested extensively both before and after being integrated into the larger satellite system. Satellite testing in general should cover vibration and thermal/vacuum testing at a minimum. More ambitious tests can be done for EMI (electromagnetic interference) and radiation tolerance, but they were beyond the scope of this effort.

4.1 Vibration Testing

Vibration testing was done on a 3-axis vibe table owned by the Mechanical Engineering department that PolySat was allowed to use. Testing was typically done by securing the item under test to a bracket, that could then be screwed directly into the table. In general a random vibration profile in each individual axis at 150% expected launch characteristics is used to provide thorough testing. Before payload design, components deemed at high-risk for vibration failure based on their connectors or geometry were individually tested. After the flight satellite is completed, a vibration test was done on the entire satellite to ensure that no components or screws came loose.

When the processor was being selected, one of the first tests was on the Gumstix and Tinyboard processors because there was concern about their connectors. Indeed, the Blackfin connector proved inadequate in the z-axis, and the Gumstix needed a custom bracket over the connector in order to survive. A metal strap mounted on the payload

board to secure the Blackfin connector solved the vibration problem for the Blackfin as well as providing a thermal sink to ensure the processor did not get too hot.

The lens housing assembly was also tested because of the possibility of the lens getting out of focus during launch by unscrewing itself from the housing. The testing was accomplished by taking a representative picture before vibe testing, and comparing that to a picture taken in the identical setting and circumstances after the vibe test. Staking the lens to the lens housing ensured that no defocusing occurred.

4.2 Thermal/Vacuum Testing

Thermal/vacuum testing is designed to test components' and materials' fitness for the space environment. The testing chamber first has all its air pumped out (the Cal Poly chamber has the capability of running at around 10^{-6} torr), and then heating tapes and liquid nitrogen are used to cycle the temperature to mimic the range of temperatures encountered in a low earth orbit. Cal Poly's satellites were typically tested between -20 and 70 degrees Celsius. It is also important to run the system during at least some of the tests in order to ensure that not only can it survive the temperature cycles, but that it can operate during them.

Thermal/vacuum testing was done first on individual components, and then on the system as a whole. The lenses were a particular concern, because of concerns that certain plastic housings or lens films could potentially outgas and cause the lenses to become less transparent. After initial testing showed that lens fogging was not a concern, a longer-term thermal/vacuum test was run on the satellite as a whole, particularly to test the payload including the imager. During this 15 hour test, the temperature was cycled

approximately two times, with images taken every ten minutes, and status data gathered every five minutes. After the test the power supply voltages and sensor current consumption were plotted with respect to temperature (see Appendix D). Power supply voltages stayed quite uniform, although current consumption varied significantly with temperature. In addition, the average black level of the images was plotted in order to ascertain the offset for a given temperature. The average black level followed the temperature cycling quite well, and a calibration offset was generated that could be used with the aid of temperature sensors near the imager to calibrate the black level of images.

4.3 Imaging System Testing

Besides the above-mentioned environmental tests, many tests were done while varying the imager settings in order to ascertain the best settings for pictures.

4.3.1 Imager Register Settings

Figure 4-1 below shows a summary of the grayscale imager register settings. The ones that were of primary concern were those dealing with exposure time since only a very small number of photons from stars would strike the imager and be converted into electricity at any given moment. In general it was necessary to turn the exposure time up significantly in order to be able to identify stars on an image. Unfortunately the images contained quite a bit of random noise that was just as bright as stars, however most major stars would appear in more than one pixel, making it relatively easy to identify.

Address	Register	Register Description	Register Values	Register Value Descriptions
05	VCLKGEN	Clock Generation	00	HCLK = MCLK
			02	HCLK = MCLK / 2
			04	HCLK = MCLK / 4
			06	HCLK = MCLK / 6
07	I2CMODE	I2C Serial Interface Configuration	AA	Standard I2C write mode
			AB	I2C advance write mode
09	OPTCTRL	Operation Control	Bit 3	Set to add extra gain for low light conditions
			Bit 2	Set to use digital video port in master mode
			Bit 0	Set to reset timing state machines
11	VSCAN	Vertical Scan Configuration	Bit 2	Enable up to down scan
			Bit 1	Enable vertical sub-sampling
			Bit 0	Enable vertical averaging
13	HSCAN	Horizontal Scan Configuration	Bit 2	Enable left to right scan
			Bit 1	Enable horizontal sub-sampling
			Bit 0	Enable horiz. averaging
15	ITIME-CONFIG	Integration Time Configuration	08	Partial frame int. on
			00	Partial frame int. off
19	WROWS	Active Window Row Start	Bits 7:0	Set start row MSBs
1A	WROWE	Active Window Row End	Bits 7:0	Set end row MSBs
1B	WROWLSB	Active Window Row LSB	Bits 5:3	Set start row LSBs
			Bits 2:0	Set end row LSBs
1C	WCOLS	Active Window Column Start	Bits 7:0	Use to program start column MSBs
1D	WCOLE	Active Window Column End	Bits 7:0	Set end column MSBs
1E	WCOLLSB	Active Window Column LSB	Bit 5	Set start column LSBs
			Bits 2:0	Set end column LSBs
20	FDELAYH	Frame Delay High	Bits 7:0	Set frame delay MSBs
21	FDELAYL	Frame Delay Low	Bits 6:0	Set frame delay LSBs
22	RDELAYH	Row Delay High	Bits 7:0	Set row delay MSBs
23	RDELAYL	Row Delay Low	Bits 4:0	Set row delay LSBs
24	ITIMEH	Int. Time High	Bits 3:0	Set int. time MSBs
25	ITIMEL	Int. Time Low	Bits 6:0	Set int. time LSBs
30	SNAP-	Snapshot Mode	Bit 5	Set for snapshot mode

Address	Register	Register Description	Register Values	Register Value Descriptions
	MODE	Configuration	Bit 4	Set snapshot pin for pulse mode, clear for level mode
			Bit 3	Set to use external shutter
			Bits 2:0	Set number of frames before image readout
31	SNAPITH	Snapshot High Int. Time	Bits 7:0	Set imaging time MSBs
32	SNAPITL	Snapshot Low Int. Time	Bits 6:0	Set imaging time LSBs
40	BLKLEV-CONFIG	Black Level Compensation	Bit 3	Disable internal black level compensation
			Bits 2:0	Adjust rate of auto black compensation circuitry convergence
41	BLK-TARGET	Black Level Target	00	Manually set black level target
42	PGA	Programmable Gain Amplifier	Bits 6:0	Set analog gain
46	OFFSET	Gain Offset	Bits 7:0	Manually set black level
83	PIXEL-OFFSET	Sensor's Pixel Offset	Bits 7:0	Compensate for natural pixel offset

Figure 4-1: KAC-9638 Imager Register Summary

Several field tests were run in order to test out the imager system involving several late night forays. Experimentation with the register settings showed that it was optimal to turn the exposure time up significantly while decreasing the gain (which tended to introduce much more noise). Successful pictures were taken on several occasions (see Appendix F for a subset of these images).

Chapter 5: Image Compression Systems

There are a large variety of image compression algorithms available, each of which has its advantages and disadvantages. Although this can be somewhat overwhelming when first approaching the problem, algorithms can be divided into a few large categories. To understand these algorithms, some background in information theory is necessary.

5.1 Information Theory [3]

While the field of information theory encompasses much more information than is necessary here, several important concepts are useful for understanding the capabilities and limitations of certain types of compression. In general, this section deals only with coding redundancy, and thus is applicable to data compression as a whole.

One of the key insights of information theory is that information itself can be quantified. Given a symbol (in image processing this is generally a pixel value) with an occurrence probability of p, the amount of information contained in the symbol is defined as:

$$I \text{ (bits)} = \log_2(1/p) = -\log_2(p),$$

when binary symbols are used. This function's graph is shown below, and demonstrates the intuitive fact that more probable events contain less "information" than less probable ones.

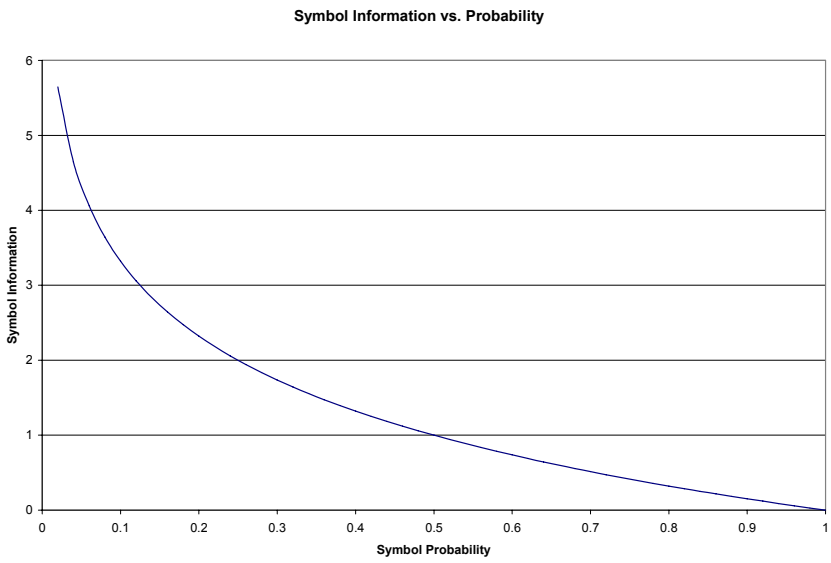


Figure 5-1: Symbol Information vs. Probability

Expanding on the above definition, the average information per symbol, or entropy of a source can be defined as:

$$H \text{ (bits)} = - \Sigma\, p_i * log_2(p_i), \text{ summed from } i = 1 \text{ to } n$$

As the following graph of the binary case shows, the entropy, or average information per symbol, is at a maximum when both symbols are equi-probable.

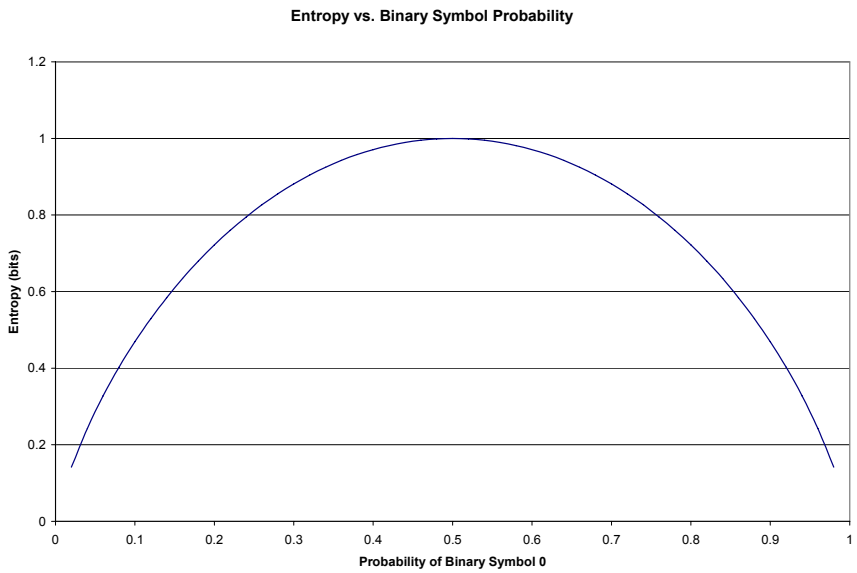


Figure 5-2: Entropy vs. Binary Symbol Probability

5.2 Basic Theory

A general compression system consists of a transform that essentially concentrates the most important information in a smaller space than the original image. This is generally followed by quantization, or some similar form of lossy compression. Because of the preceding transform, the loss should be almost unnoticeable in the restored image. Finally the image is variable-length coded, using Huffman or a similar code. Other methods can be used to supplement or replace portions of this process. For example, dictionary coding schemes such as LZW, often provide very efficient results for lossless compression.

5.2.1 Coding Redundancy

On a conceptual level, coding redundancy describes the fact that, when an information source produces symbols with non-uniform probability, coding the more

common symbols with smaller number of bits, and the rarer ones with larger numbers, decreases the overall amount of memory needed to store the same information. This type of coding is called variable-length coding, and is not specific to image files, but works identically on any type of data.

The amount of compression theoretically possible by eliminating coding redundancy can be found through a calculation of the source's entropy. Entropy is the average amount of information gained through observation of one sample (in this case specified in bits/pixel). Here, it can be used as a lower bound on how many average bits are theoretically needed to code each pixel. For a uniform source, there is no coding redundancy, but the more varied the probabilities are, the more savings are possible. If r_k is a discrete random variable in the interval [0,1] that represents the gray levels of the image, then the average number of bits required to represent each pixel in the image can be calculated as:

$$L_{avg} = \sum l(r_k)p(r_k), \text{ summed from } k = 0 \text{ to } k = L\text{-}1 \text{ [3]},$$

where $l(r_k)$ is the number of bits used to represent each value of r_k, $p(r_k)$ is the probability of each r_k, and L is the total number of gray levels. Then the total number of bits required to represent an image is equal to:

$$M * N * L_{avg} \text{ [3]},$$

where M is the number of rows in the image, and N is the number of columns. A lower bound on the compression ratio possible by eliminating coding reduction can be calculated using the following formula as:

$$M * N / M * N * L_{avg} = 1 / L_{avg}.$$

5.2.2 Interpixel Redundancy

Also known as spatial redundancy, interpixel redundancy indicates the amount of correlation between adjacent pixels. The main difference between image compression and other types of digital data compression is the amount of spatial redundancy in an image. Although sensor readings that need to be compressed may show repetitive patterns over time, an image has two-dimensional redundancy that can be used in predictive type algorithms. The average picture has vast areas of constant, or nearly constant colors, with few abrupt changes. Huffman encoding, an entropy reduction algorithm, while optimal in certain respects, does not take spatial relationships in the data into account.

The autocorrelation of an image is a measure of how much the image varies over its two dimensions. For images of stars, which have a large amount of autocorrelation because of the numerous black areas, spatial redundancy reducing algorithms can be very beneficial.

5.2.3 Lossy vs. Lossless

Most image compression systems incorporate some kind of lossy compression since even a small amount of loss can achieve dramatically smaller file sizes. Most loss is incurred in a quantization stage, where the least significant bits are truncated in order to reduce the file size.

In some situations, although lossy image compression can be quite good, lossless compression is a requirement. For example, when used in medical imaging, lossless

compression is often required. Space applications are another case where it is felt that the importance of getting all the data overweighs the desire to compress the image.

In this thesis, although an argument could be made for lossy compression, the decision has been made to focus on lossless compression. This will ensure that the capabilities and limitations of the CMOS imager are better understood, as well as ensuring that the images received on the ground are identical to those used in on-board image processing.

5.2.4 Multidimensional Resolution

Multidimensional resolution allows the most significant portions of an image to be compressed separately from the least significant portions. In many cases, the most significant portions can have a much higher compression ratio because the least significant portions have a large noise component.

Multidimensional resolution is often preferred in situations such as downloading images from the internet. In this case, the entire image, albeit fuzzy, is available quickly, and the detail is successively refined as more information is received. In downloading images from space, this ability is also desirable, since the first approximate image could give us enough information to know if the whole picture is worth downloading or not.

5.2.5 Channel Coding

Channel coding in some senses is the opposite of source coding. While source coding is designed to make the smallest number of bits convey the largest amount of information, channel coding is used to introduce more redundancy into the information, so that if error is introduced in the transmission channel, it can be corrected. Because

many of the algorithms compress the information significantly, even a change in a single bit can cause the image to become decompressible. Most satellite communication systems have built in error detection/correction, which somewhat mitigates this problem, but certain algorithms are more robust to bit errors.

5.3 Compression Quantification and Comparison Parameters

The main performance metric is compression ratio. Because only lossless compression is considered, any error measures are unnecessary. Besides this quantitative metric, several other qualitative comparison parameters should be considered. These deal with how easy the system is to implement in a CubeSat.

5.3.1 Quantitative

As mentioned above, the main metric is compression ratio, which is simply the ratio of the size of the uncompressed file to the compressed file. A maximum compression ratio can be calculated in terms of the entropy of the source. This essentially indicates how much the file could be compressed if only entropy reduction methods (such as Huffman coding) were to be used. As such, spatial redundancy reductions can exceed that compression ratio. Still it is a useful measure for calculating whether a Huffman or arithmetic code will be able to compress a file very much.

In addition to this, although the amount of time required to compress an image varies with processor, relative compression times can be calculated for the different schemes. All other things being equal, a shorter time is desirable.

Memory requirements also vary according to the scheme. There are two components to this. First, the amount of memory required to store the code should be as

small as possible. Second, it should use as little memory as possible when compressing the image.

5.3.2 Qualitative

One qualitative consideration to take into account is the compressed file's susceptibility to packet loss and/or errors introduced by a noisy channel.

For packet loss situations, multidimensional resolution methods are preferred, since most of the detail of the entire image is downloaded in the first packet(s). Thus, even if future packets are lost, a fuzzy version of the image is available, even if not shown in complete detail.

Noisy channels are a serious concern because of the possibility that a bit error introduced at one point could make the entire file unable to be decompressed. The CRC check used by the Cal Poly CubeSats makes the possibility of a bit error reasonably small. Nonetheless, algorithms that allow decoding in the presence of dropped packets or flipped bits are preferable.

5.4 Types of Image Compression

This section contains a summary of many of the most common types of image compression algorithms, along with their advantages and disadvantages. More detailed descriptions of algorithm implementation can be found in Chapter 6. The information in this section came from the image and data compression references [4], [7], [12], [13], [14], [15], and [16] cited in the references section below. While the list is not exhaustive, the following image compression techniques are among the most popular and widely-used.

5.4.1 Transforms

A transform simply translates data from one domain, such as time, to another, such as frequency. Transforms are inherently lossless and reversible, and are simply an alternate way of representing the same information. Although transforms can be used on any type of compressible data, they are particularly useful for images, especially those with a fair amount of spatial redundancy.

Although transforms are inherently lossless, they are generally only used in lossy compression (with the exception of JPEG2000—see below). That is because transforming an image does not generally reduce its size. Size reduction comes from the quantization of the least significant bits (see quantization section below). By transforming and then quantizing, only the least important information is lost.

The primary advantages of transforms is the ability to tailor them to best preserve the features of most interest. A disadvantage is that they are computationally costly.

5.4.1.1 Fourier and Discrete Cosine Transforms

One of the most common transforms is the Fourier, which converts from the time to frequency domain. For an image, a 2-D transform is used. If the image does not have any sharp edges or high contrast areas, most of the information is concentrated in the low frequencies, giving an image with most nonzero values near the origin. Although there is no theoretical reason a Fourier transform couldn't be used in compression techniques, in practice other methods more amenable to computer manipulation are generally used. For example, the discrete cosine transform (used in JPEG compression) avoids the problem of imaginary coefficients by only using cosines.

5.4.1.2 Wavelet Transforms [6], [16]

The hottest area of development is currently in wavelets, which are time-limited waves of various forms. By limiting the duration of the wave, it better matches sharp contrast areas, as opposed to a sine wave, which persists indefinitely in time. For example, it takes sine waves at an infinite number of frequencies to match a vertical edge, where a wavelet could do it with many fewer. Thus wavelets are ideally suited for images with high frequency components.

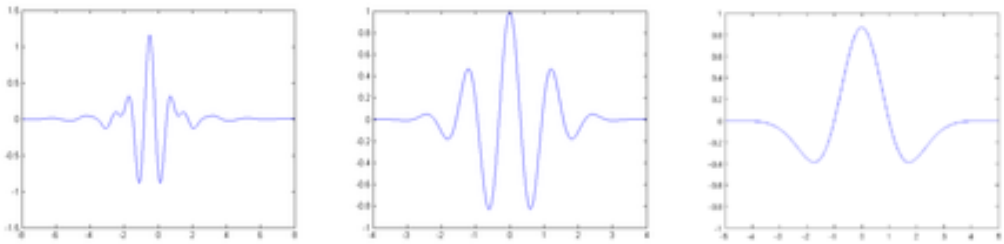


Figure 5-3: Sample Meyer, Morlet, and Mexican Hat Wavelets [20]

Another way of thinking of wavelets is in the frequency domain. Whereas a Fourier transform has constant bandwidth in the frequency domain, a wavelet has relative constant bandwidth. This means that a longer time (or spatial, for images) window is used for lower frequencies, whereas a shorter time window is used for higher frequencies. Thus the wavelet transform allows for more efficient bit allocation for higher and lower frequency areas.

5.4.2 Entropy Encoding

Entropy encoding uses (generally variable length) codes that look only at the frequency usage of various words or pixels in the data set. Because they do not consider the positions of the words or pixels, these codes take no account of spatial redundancy,

and work identically on a picture and on a random data set. Essentially they work by encoding common words or pixels in fewer bits than uncommon words or pixels. There is a lower bound on the compression ratio based on entropy encoding (see section 5.1.1 above).

5.4.2.1 Huffman Coding

Huffman coding is a widely used method that has been proven to be the code that closest approaches the entropy bit rate (using codewords with integer numbers of bits). It is a universal variable length code, which means that each code word is uniquely decodable with a dictionary when scanning through the text. It is often used as the final step of a compression scheme, after a transform or other compression algorithm has first exploited the image's spatial redundancy.

One of the main advantages of Huffman coding is that it is instantaneous and uniquely decodable. Unfortunately, a true Huffman code requires knowledge of the source statistics, which must then be included in the transmission of the compressed file so that the dictionary can be built up on the decoding side. However there are other forms of Huffman coding that do not require a priori knowledge of the source statistics—they are not optimal, however. In addition, because Huffman coding does not exploit spatial redundancy, by itself it often provides less compression than those methods that do.

Other near optimal codes may be more useful in some situations if they are computationally simpler. They are not considered here, however, because, the a priori knowledge of source statistics is possible for this imaging system.

5.4.2.2 Arithmetic Coding

Arithmetic coding is another entropy encoding method. It actually manages to achieve better performance that more closely approaches the theoretical bound, than the Huffman encoder because it does not require that each source symbol be mapped to an integral number of bits.

The basic approach is to begin with a range from 0 to 1, and for each successive symbol to narrow the range in a manner corresponding to its relative probability. The disadvantage of this method is that it requires a fair amount of scaling and rounding to avoid problems caused by finite precision arithmetic.

5.4.3 Quantization [6], [16]

Quantization is usually the only nonreversible part of an image compression scheme, and provides the most compression by introducing loss of the least significant bits, whether of an uncompressed or previously compressed image. Although a standard quantizer can be used, better performance is often achieved by introducing a nonuniform quantizer.

5.4.3.1 Uniform Quantization

Uniform quantization, while often not as effective against nonuniform input distributions, is often the simplest scheme to implement. Uniform quantizers are defined by the fact that both the x-axis (input) intervals and y-axis (output) intervals have uniform spacing except at the endpoints. Given these constraints, either a midtread or midrise quantizer is possible. Uniform quantizers are optimal (minimum mean square error) only when the input probability distribution is uniform.

5.4.3.2 Nonuniform and Adaptive Quantization

Although more complicated to implement, nonuniform quantization is more optimal when the input distribution has a nonuniform probability distribution. Details on how to define a quantizer to minimize the mean square error can be found in several of the references [14], [15].

If the input statistics are not well known, or are time-varying, optimal quantization can only be achieved with an adaptive quantizer, which uses adaptive filtering methods to adjust the quantizer's decision levels to the statistics of the input sequence. Naturally, this is much more complex to implement than either a uniform, or fixed nonuniform quantizer.

5.4.4 Predictive Coding

Predictive coding can be either lossless or lossy, depending on if the error is quantized or not. At a basic level, a predictive coder calculates the likely next value of a pixel based on previous value(s). Then the error is calculated as the difference between the actual and predicted value. Instead of the actual pixel value being stored, only the error value is stored (or a quantized version thereof). If a good predictive algorithm is employed, the error should be quite small, and vary little from pixel to pixel, especially if there is a large amount of spatial redundancy.

The simplest method is simply to use the previous pixel value as the estimate of the next. Even this can substantially reduce the amount of information needed to encode the picture.

Of course, although the values are typically smaller, predictive coding does not in and of itself reduce the size of the file. Instead the error values must then be coded using some variable coding scheme or dictionary method. In general, the maximum compression of a lossless approach can be estimated by dividing the average number of bits used to represent each pixel in the original image by a first-order estimate of the entropy of the prediction error data [4]. In addition the predictive process reduces the amount of interpixel redundancy, meaning that dictionary methods, etc. will not work as well on it.

Thus predictive coding is most commonly used with quantization of the error, providing a lossy image compression scheme (Delta Modulation). An optimal predictor is designed to minimize the encoder's mean square prediction error.

5.4.5 Dictionary Methods [6], [16]

Unlike previous compression techniques based on a probability model of the input source, in dictionary methods a symbol or string of symbols from the source alphabet is represented by referencing an entry in a dictionary constructed from the source alphabet. These methods work best when frequently occurring combinations of source symbols occur together.

Dictionary compression methods can in general be either static or adaptive. Static methods will provide the best compression ratio if delay is not a significant concern. Adaptive coding must be used if a completely defined dictionary is impossible to generate before the encoding process. In this case the dictionary changes based on the input during the coding process.

5.4.5.1 LZW Coding

While there are several dictionary-coding methods, LZW is by far the most common dictionary coding method in use, because it overcomes many of the disadvantages of other methods.

As the image is scanned line by line, a dictionary is built with sequences of words of ever-increasing length that appear in the image. Then only the dictionary address values are coded, which can provide large savings in the number of bits used. The downside is that, for very random sources, this type of compression can actually increase the size of the file.

This is a very popular method. It works best for images or data with a high amount of redundancy. It is used in the GIF, TIFF, PDF, and ZIP formats. One of its chief advantages is that no dictionary need be transmitted with the compressed image. Instead, the dictionary can be recreated as the same process is repeated on the decoder side.

5.4.6 Bit Plane Encoding

While not actually a compression method, dividing an image up into bit planes, with each being a binary representation of the most significant through least significant bits of each pixel, can provide ways of better compressing each plane. By looking at the characteristics of each plane, a compression scheme can be designed for that plane in particular, allowing it to be optimized individually. For example, the most significant planes will have the most identifiable pattern, and thus benefits from a spatial redundancy method. The lower planes however, will appear virtually indistinguishable from noise,

and are best encoded using Huffman or arithmetic coding. Figure 5-4 below shows a grayscale image broken up into its individual bit planes, from most to least significant.

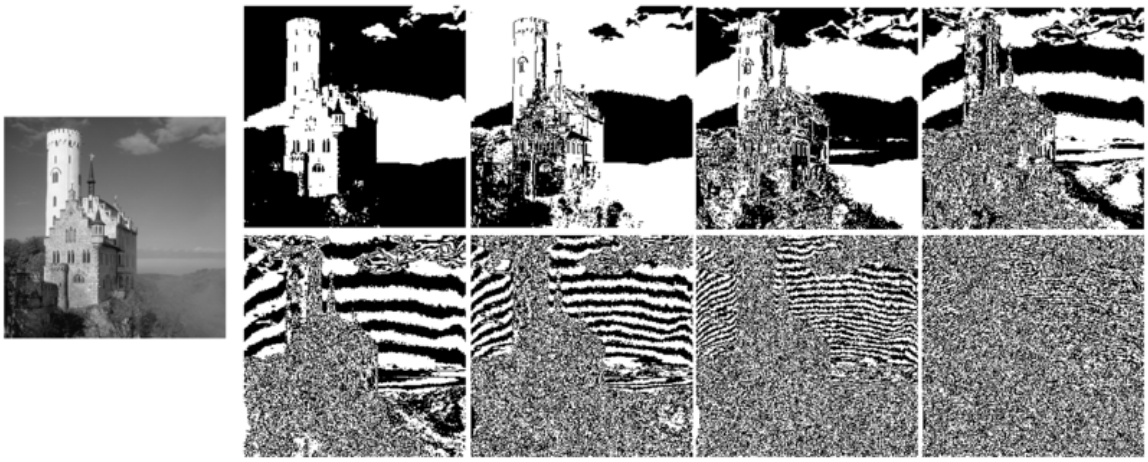


Figure 5-4: Sample 8-Bit Image Divided into Bit Planes [18]

Gray coding can also improve this method. A gray code is a binary code where each value differs from the next by only one bit. Thus, instead of the transition from, for example, 127 to 128 requiring all eight bits to change, when gray coded only one bit changes. This makes the bit planes more uniform and thus more compressible.

The main disadvantage of this method is that it is very dependent on the type of image taken. That is, the compression method best suited for each plane will vary highly with the type of image. In addition, it requires a fair amount of computation, because of the different methods used for each plane. Also, there must be enough header information to allow the receiving computer to accurately reconstruct the image.

5.4.6.1 Run-Length Coding [6], [16]

Run-length coding is among the simplest binary coding technique. Essentially the number of contiguous black pixels in a row is coded, followed by the number of white

pixels, etc. Clearly this method is most advantageous when there is a high amount of spatial redundancy in the image. In 2-D run-length coding, the two-dimensional spatial redundancy of an image is taken into account.

5.4.7 JPEG Compression Standards [6], [16]

JPEG is probably the most commonly used lossy image compression standard. Although JPEG2000 was designed to supercede it, the adoption process is still in place. JPEG compression is based on the discrete cosine transform. The transformed image is then quantized, and finally Huffman encoded. Although JPEG works quite well, it suffers from blockiness at high compression ratios because of the difficulty of representing high frequency transitions with cosines. A lossless coding scheme based on predictive coding is also part of the standard, though little used. Progressive coding is also available in the hierarchical mode for those applications that require multiresolution analysis. This is done by downsampling the image at various frequencies.

JPEG2000 is the new image compression standard from the Joint Photographic Experts Group that brought us the original JPEG standard. It relies upon the same basic format as JPEG compression, that is, a transform, followed by quantization and entropy coding. However, because a discrete wavelet transform is used instead of JPEG's discrete cosine transform, increased compression ratios and higher fidelities can be obtained. A lossless version using the integer wavelet transform is also available.

Chapter 6: Compression System Development

6.1 Test Images

The set of test images is comprised of pictures taken of the sky at night with the engineering model satellite. The test set contains images taken with a variety of exposure times and gain settings typical of the application. The images were compressed with the algorithms as detailed below. Image sizes ranged from 340 x 340 up to the full 1024 x 1240 resolution, although most images are 512 x 512. The set of test images can be found in Appendix F (contrast is adjusted to show detail).

6.2 Software

The candidate compression methods were implemented in C, which is the primary non-assembly programming language for microprocessors. Although each microprocessor may have slightly different versions of C, standard C code will be relatively easy to port to different microprocessor platforms.

6.2.1 Environment

Visual Studio and Visual C++ Express were the development environments used for this project. They were chosen because of their easy user interfaces. A few standard C libraries were used, but all compression functions were custom written.

6.2.2 Algorithms

Because the goal of this thesis was to compare the performance of different compression systems, several different algorithms were implemented in C, and their

relative performance analyzed. Instead of using already available C libraries with compression functions, the decision was made to write the algorithms from scratch. This was done for two reasons. First, it allowed for a greater understanding of how the compression algorithms worked than simply using someone else's code. Second, many of the libraries do not allow as much variation in the details of how the algorithms are implemented.

6.2.2.1 Algorithms Not Used

As mentioned above, lossy methods were categorically not considered for this thesis. Only entirely reversible compression methods were implemented.

In general, transform methods were not considered because they are usually lossy. The one exception is the discrete integer wavelet transform used by JPEG2000, which provides lossless compression. While not specifically coded, an Adobe Photoshop plug-in was used to convert the test images to J2000. Because their compression ratios were not any better than the algorithms implemented in the compression program, and the computational requirements of this method were large, it was not pursued further.

Arithmetic coding was not implemented because of its relative difficulty of implementation and only slight advantage over Huffman coding. Since other methods were able to exceed the compression ratio possible by entropy encoding anyway, it did not seem worthy of further pursuit.

6.2.2.2 Algorithms Implemented in Compression Program

The following techniques were implemented in C and used to compress and decompress each image of the set of test images.

Huffman coding was the first method implemented. Compression ratios were typically between 2:1 and 7:1 (results for all techniques are detailed in Section 6.5 below, and in Appendix G).

LZW coding was the next method implemented. Results varied widely based on image characteristics. In general, as exposure time was increased and more noise was introduced into the image, Huffman coding performed as well or better than LZW. However, for those images that were quite dark, LZW was able to achieve compression ratios as high as 320:1. Because Huffman encoding tends to destroy interpixel redundancy, while LZW coding results in a compressed file with a large amount of entropy, therefore using the methods sequentially did not achieve noticeably better results and was not included in the final test sequence.

Predictive coding followed by Huffman and LZW coding was implemented next. Generally speaking, predictive coding plus Huffman coding or LZW coding did not result in as high a compression ratio as Huffman or LZW coding alone.

The final technique attempted was bit plane coding. Each image was broken up into individual bit planes, one comprised of the most significant bit of each pixel, the next comprised of the second most significant bit of each pixel, etc. Because each plane can be individually compressed, this technique allows for the compression method to be targeted to the characteristics of each bit plane. Each of the bit planes was then individually compressed using either LZW, Huffman, or non-zero encoding. In general, results showed that breaking the image into bit planes and compressing did not result in compression ratios greater than those achieved by Huffman or LZW on the whole image.

The exceptions were those images with low exposure times that fared well under zero-based RLC encoding, resulting in compression ratios greater than 2500:1.

6.3 Compression Algorithm Program

The above-mentioned techniques were carried out in sequence on each image from the set of test images. The program code performed as follows in sequence:

Huffman & LZW encoding:

- Read in image file
- Perform compression technique on image
- Write compressed file
- Read in compressed file
- Decompress file
- Compare to original file
- Record compression statistics

Predictive encoding:

- Perform predictive coding on image and generate predictive array
- Perform Huffman & LZW techniques on predictive array
- Write compressed files
- Read in compressed files
- Decompress files
- Compare to original files
- Record compression statistics

Bit plane encoding:

- Decompose image into 8 bit plane images

- Perform Huffman, LZW, and zero-based RLC techniques on each bit plane image
- Write compressed files
- Read in compressed files
- Decompress files
- Compare to original files
- Record compression statistics

The above sequence was repeated for each image.

6.4 Algorithm Details

The following algorithms were implemented in the compression program in sequence to test out the various compression methods on the test images. Following are the details of how each algorithm was implemented. This section is largely based on references [4], [7], [12], [13], [14], and [15].

6.4.1 Huffman Coding

In Huffman coding, a Huffman tree is generated based on the image statistics. The first step in constructing this tree is to generate a histogram of the image and to calculate the probabilities of each pixel value based on that histogram. Those probabilities are then listed in descending order as shown in an example in Figure 6-1 below.

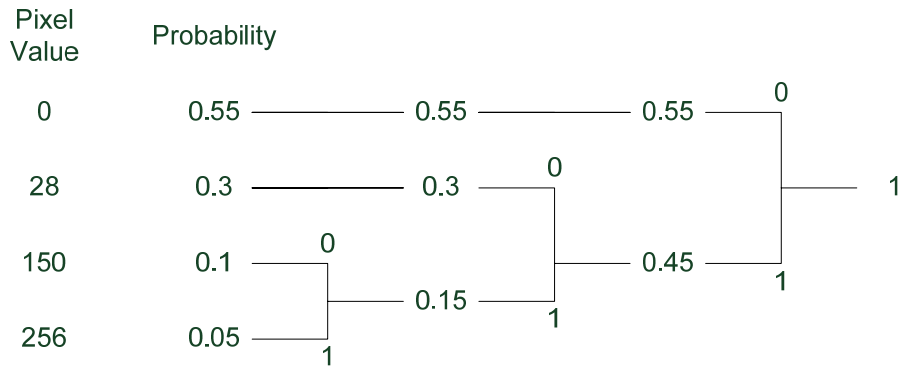


Figure 6-1: Huffman Tree

Reading the Huffman tree from right-to left, the least common pixel value is assigned a 1, and the second least common pixel value a 0. Then their probabilities are added together and carried forward to the next step of the process. The next two least common probabilities are again assigned a 1 and a zero, and their probabilities again added together. This process is repeated until all probabilities have been summed together. Then the code is generated for each pixel by reading the 1s and 0s backwards (from right to left) until the pixel value is reached. Thus, the variable length code table.shown in Figure 6-2 is generated.

Pixel Value	Probability	Code
0	0.55	0
28	0.3	10
150	0.1	110
256	0.05	111

Figure 6-2: Huffman Coding Table

As Figure 6-2 shows, Huffman codes are variable length codes that code relatively probably pixel values with smaller numbers of bits. Of course, the code must be transmitted along with the data so that the image can be decompressed at the other end.

6.4.2 LZW Coding

LZW coding uses interpixel redundancy to reduce data size. As the program looks through the image pixel by pixel, a dictionary is generated with common combinations of pixels represented by one eight-bit value. A sample LZW code is generated in Figure 6-3 below.

24 96 24 24 24 96 96 24 96 24 24

Pixel Input	Code Output	New Dictionary Index	New Dictionary Value
24 (96)	24	256	24-96
96 (24)	96	257	96-24
24 (24)	24	257	24-24
24-24 (96)	257	258	24-24-96
96 (96)	96	259	96-96
96-24 (96)	257	260	96-24-96
96-24 (24)	257	261	96-24-24
24	24		

Figure 6-3: LZW Coding Table (Sample Data at Top)

For an 8-bit image, the first 256 dictionary entries are 0 through 255, and the indices and values are the same. When compressing the image, pixels are read in sequence until an unknown sequence is encountered. As shown in the example above, 24 followed by 96 is read, for which there is no dictionary entry. Therefore 24 is output , and a new dictionary entry at index 256 is created with the 24-96 sequence. The next time that sequence is encountered, the shorter index 256 can be used rather than 24-96. This process continues until the image is finished. For short or very random files, LZW offers little, and sometimes a negative compression rate, but for files with a lot of redundancy, it is very powerful. One of its strengths is that the dictionary code does not need to be sent along with the compressed file. Instead, the code can be regenerated in the same way by going through the compressed file in sequence and creating dictionary entries.

The compression program used mostly standard LZW coding with a few exceptions. Normally dictionary entries 0 through 255 are pre-populated with the values 0 through 255. It was found that a higher compression ratio in the mostly dark star images was possible by generating the dictionary completely from scratch. The only downside was that the image histogram had to be sent along with the compressed file. The other change had to do with the dictionary size. Normally a fixed number of dictionary entries is allowed, and the number of entries determines the number of bits per symbol in the code. In the compression program, utilizing a variable length code where the number of bits per symbol in the code increased as the dictionary size increased further reduced the size of the compressed file.

6.4.3 Predictive Coding

In predictive coding, an array of values is generated based on the difference between each pixel and the one before it. If the image has a lot of interpixel redundancy, these numbers should be small, and therefore able to be compressed more easily. In the compression program, predictive coding is followed by Huffman and LZW compression for comparison.

6.4.4 Bitplane Coding

In bit plane coding, an 8-bit image is broken up into eight 1-bit images. For most images, the planes containing the most significant bits will be almost uniform, while the least significant planes are almost random.

In the compression program, the image was first broken up into individual bit planes and then separately compressed each of the planes using three methods: Huffman coding, LZW coding, and zero-based run-length coding. In zero-based run-length coding, a length of zeros is represented by a 0 followed the number of zeros in the run. For the most significant planes, this reduced the size to almost nothing in star images.

6.5 System

While developing a complete system was beyond the scope of this thesis, a functional block diagram is included to show how a compression system would likely be integrated into a satellite communications system.

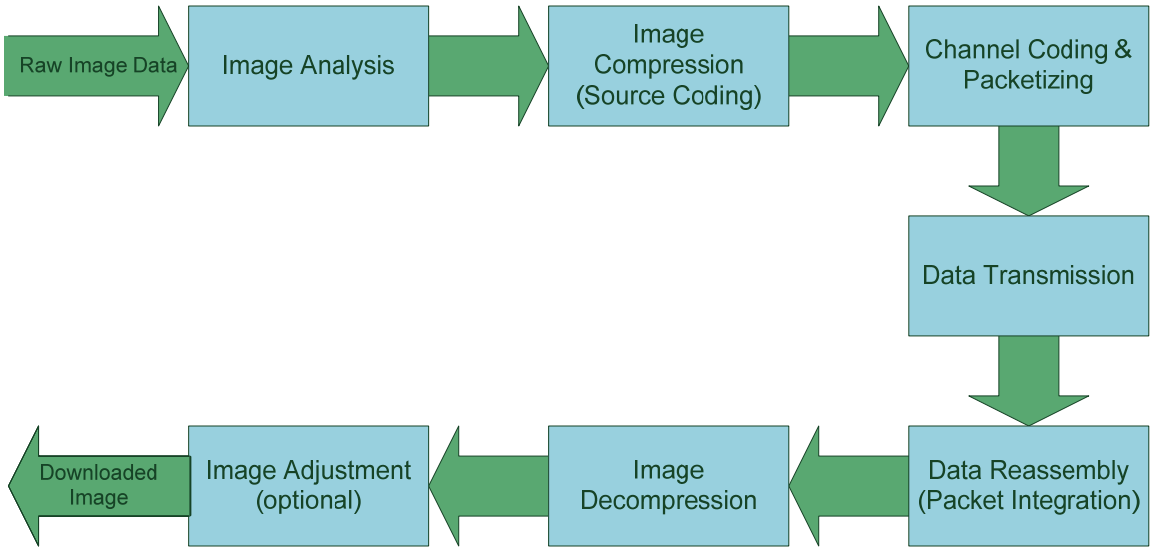


Figure 6-8: System Block Diagram

The raw image data would first undergo basic image analysis to determine the image's entropy and standard deviation at a minimum. If desired or necessary, the analysis software could even determine whether the image was of the sun, moon, stars, or earth. Regardless, data from the image analysis would be used to choose among a variety of compression techniques. After image compression, the data would undergo channel coding and packetizing. In choosing the channel coding algorithm, a trade-off would have to be made between robust error-checking and smaller data size. After data transmission to the earth station, individual packets are reassembled into the complete data file, which can then undergo image decompression. The final step is image adjustment (if desired) to compensate for overly short or long exposure times or other image defects.

Chapter 7: Conclusions

7.1 Imaging System Hardware

During the course of this thesis, an imaging system was built from scratch, from part selection through schematic design, board layout, population, and finally test. Although the final system was successfully able to take images, the hardware did not always work as seamlessly as desired. If given the chance to do things differently, the author would have put more time, effort, and documentation into the initial design and board layout in order to avoid some of the problems that occurred later.

For example, the imager was selected partially because of its high sensitivity, which we deemed necessary for high-quality star images. However, further ground-based testing has shown that the combined effects of the imager and lens have resulted in an imager system that has *too* high a sensitivity to light, and does not take good pictures in daylight, making it problematic for use in earth and sun pictures.

As CubeSat designs increase in ambition and capability, more teams will likely decide to create their own custom imaging system. It is the author's hope that this thesis can provide some assistance in that task.

7.2 Image Compression Algorithms

The results of the Compression Algorithm Program for each of the 22 test images are summarized below and complete results are included in Appendix G.

Image Content	Size (resolution, bytes)	Entropy	Mean, Standard Deviation	Optimal Compression Scheme	Compressed Size (bytes, compression ratio)
moon & stars	1024 x 1280 1,310,720	3.91	12.28 9.70	Predictive + Huffman	625,766 2.1:1
star	512 x 512 262,144	3.10	41.60 2.94	Huffman	102,709 2.6:1
star	512 x 512 262,144	1.60	9.74 0.77	Huffman	56,280 4.7:1
appears empty	512 x 512 262,144	2.52	11.23 2.93	LZW	69,838 3.8:1
Orion	512 x 512 262,144	3.77	17.86 4.10	Huffman	75,973 3.5:1
Orion	512 x 512 262,144	2.57	12.60 1.77	Huffman	37,519 7.0:1
Orion	512 x 512 262,144	2.32	10.21 1.64	LZW	60,787 4.3:1
star	512 x 512 262,144	0.002	0.002 0.23	Bit plane + Zero RLC	200 1310.7:1
star	512 x 512 262,144	2.26	7.63 2.53	LZW	56,752 4.6:1
Orion	512 x 512 262,144	2.36	13.78 3.09	LZW	36,722 7.1:1
Orion	512 x 512 262,144	2.36	13.78 3.09	Huffman	58,182 4.5:1
star	512 x 512 262,144	0.001	0.001 0.12	Bit plane + Zero RLC	100 2621.4:1
possible stars	512 x 512 262,144	2.02	12.33 3.05	LZW	51,698 5.1:1
star	1024 x 1280 1,310,720	2.99	16.41 4.04	LZW	483,940 2.7:1
star	1024 x 1280 1,310,720	3.18	7.68 4.59	Huffman	528083 2.5:1
Orion	1024 x 1280 1,310,720	1.32	3.14 0.61	LZW	225263 5.8:1
Orion	1024 x 1280 1,310,720	2.77	2.32 2.01	Huffman	462875 2.8:1
stars	1024 x 1280 1,310,720	1.48	3.15 0.70	LZW	253840 5.2:1
moon glow	512 x 512 262,144	6.15	88.86 20.24	Predictive + Huffman	145977 1.8:1
moon glow	512 x 512 262,144	3.88	18.50 4.21	Predictive + Huffman	114614 2.3:1
moon	512 x 512 262,144	5.48	42.11 49.34	Predictive + Huffman	72801 3.6:1
partial moon	512 x 512 262,144	5.43	53.00 27.97	Predictive + Huffman	101638 2.6:1

Figure 7-1: Compression Test Results Summary

As Figure 7-1 shows, the optimal compression scheme was highly dependent on image characteristics such as information content (entropy), image brightness (mean pixel value), and noise (standard deviation).

The following figures show details of plots of compression ratio vs. image number, entropy, and standard deviation (full plots are included in Appendix G). Figure 7-3 shows a clear correlation between decreasing entropy and increasing compression ratio, while Figure 7-4 shows a similar trend between decreasing standard deviation and increasing compression. However in Figure 7-4, the correlation is not as clear.

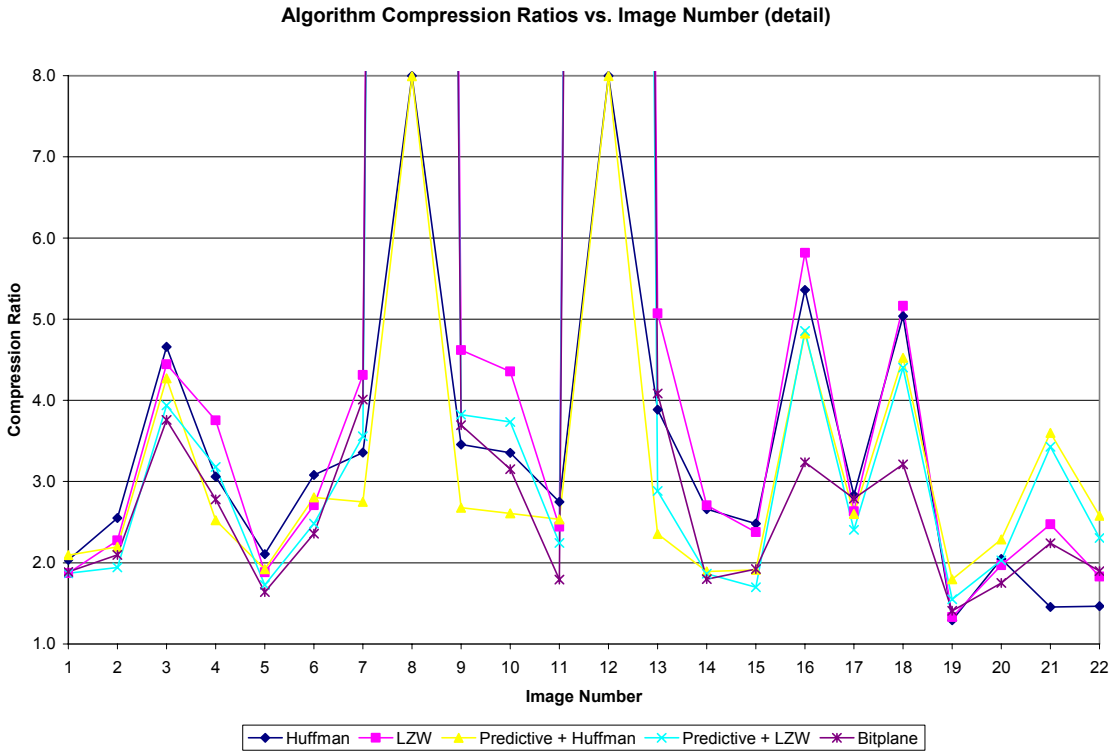


Figure 7-2: Compression Ratio vs. Image Number (detail)

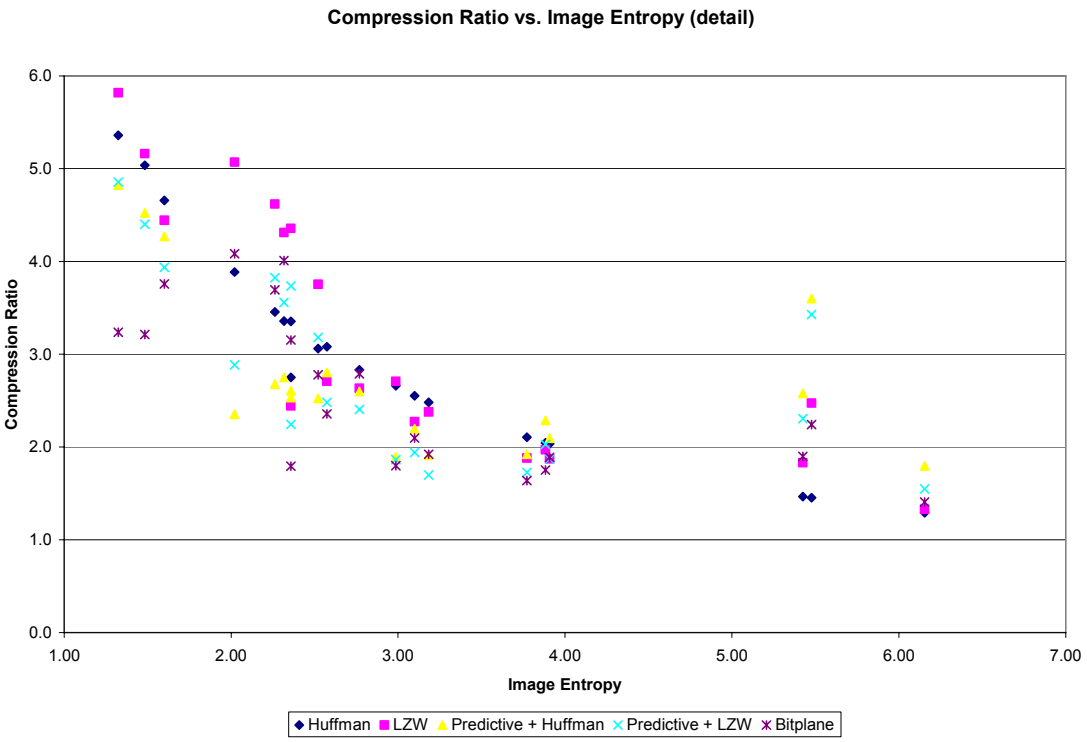


Figure 7-3: Compression Ratio vs. Image Entropy (detail)

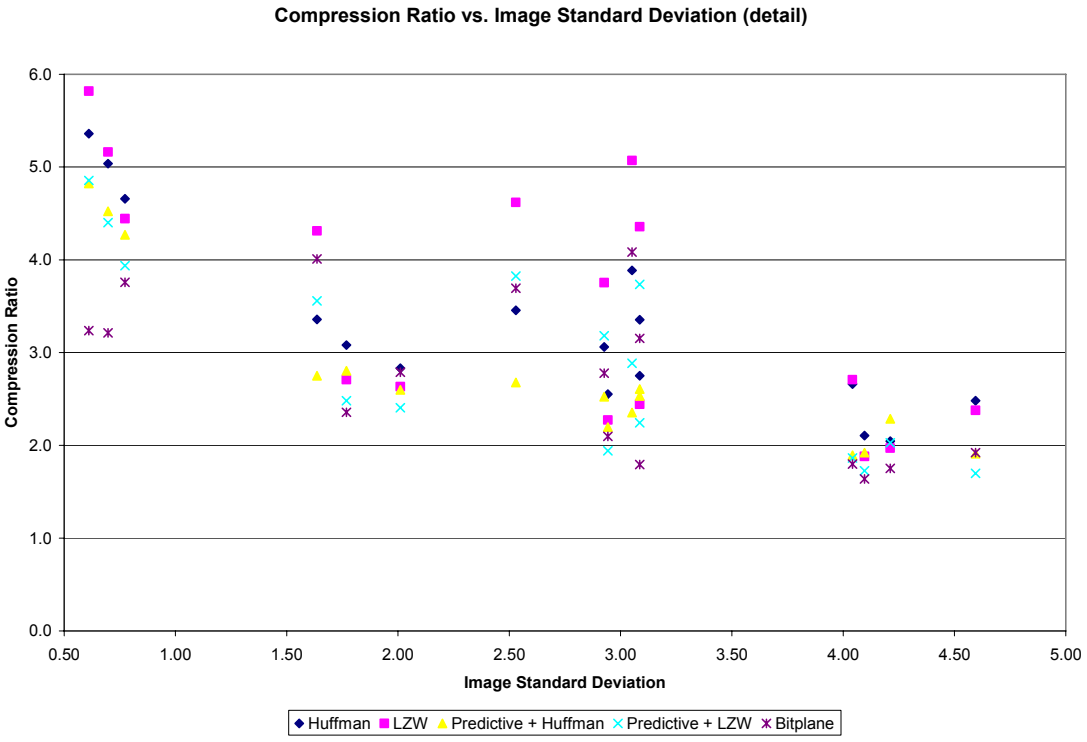


Figure 7-4: Compression Ratio vs. Image Standard Deviation (detail)

Based on the dependence of the optimal compression scheme on the image statistics, it is clear that a simple image analysis would be advisable before compressing each image. General guidelines could be formulated to relate the image's entropy and standard deviation to the choice of compression scheme, potentially saving a significant amount of downlink time. From the above analysis, it appears clear that images with lower entropy values should be compressed using LZW. An image analysis may also determine whether the image depicts the sun, moon, earth, or stars. Each of these types of images may be preferentially encoded using differing methods.

One significant downside to using LZW coding is the requirement that every packet be retrieved before decompression can take place. For this reason, depending on the quality of the satellite-ground station link, LZW may be undesirable for some systems.

7.3 Future Work

Although this thesis will provide a good start for space image compression, there is much more that could be investigated. Specifically, this thesis focused on grayscale image compression, but RGB images provide other opportunities for investigating alternate compression methods. In addition, images that contain the sun or the earth may be better compressible using other methods. An optimal image compression scheme would first identify the type of picture, and then apply the corresponding algorithm. The compression techniques documented above would need to be implemented on an embedded system, which would likely require some programming rework and optimization.

The overall throughput of the system is severely limited due to the low-speed data buses and problematic satellite-to-ground communications. Improving those links would significantly improve the utility of imager systems on CubeSats.

Chapter 8: Lessons Learned and On-Orbit Results

8.1 Lessons Learned

There are numerous registers on the imager, many of which do not need to be changed. Therefore, the payload software only made provision for changing some registers. In retrospect, it would have been preferable to be able to write new values to some of the unchangeable registers. In particular, the image size was fixed, which made it impossible to use the option of taking a smaller image to limit the light collected by the imager.

On a more general note, the payload software architecture was rather ambitious as it was intended to enable more functionality than the team ended up having time to accomplish. Some members of the software team in retrospect would have made the software simpler so that there were fewer bugs and they could have spent more time testing it.

Several board layout errors complicated matters. The most problematic was the miswiring of the vsync and hsync pins which led to complications for the DMA image transfer. Further verification of the hardware layout would have prevented this problem.

Significant difficulty was found using the parallel data port on the imager. It was running at 15 MHz, substantially faster than any other processor or chip on the satellite. The author's unfamiliarity with high data-rate systems led to mistaken assumptions that signal propagation would only be quantitatively different at the higher speeds. Instead, the high frequencies led to many unexpected and intermittent problems. The lack of a

sufficiently high bandwidth oscilloscope made troubleshooting these problems quite difficult.

8.2 On-Orbit Results

CP-3 was still under development when the author left Cal Poly to take a full-time job. As such, she was not fully involved with the final software development and satellite integration. Ultimately, because of unsatisfactory tri-state buffering between the two imagers, only the black-and-white imager was populated and functional on the flight version of CP-3. LZW compression by a software team member (David Cuddeback) was implemented on the payload.

Unfortunately, the communications link between the satellite and ground has been unreliable, and only a few commands have gotten through. The satellite has been commanded to take a picture, and it was stored. However, attempts at downloading the image were unsuccessful, and it has not been possible to take another image.

CP-6, currently under development, is substantially similar to CP-3, but with an improved satellite bus and payload hardware. In particular, the improved communications system should allow for image download, and verification of payload hardware and software.

LIST OF REFERENCES

1. Aalborg University's AAU CubeSat website, URL: http://www.cubesat.auc.dk/.
2. CubeSat website, URL: http://www.cubesat.org.
3. Golomb, Solomon W., *Basic Concepts in Information Theory and Coding: The Adventures of Secret Agent 00111*, Plenum Press, New York, 1994, Ch. 2-7.
4. Gonzalez, Rafael C. and Richard E. Woods, *Digital Image Processing*, 2nd edition, Prentice-Hall, Upper Saddle River, NJ, 2002, Ch. 7-8.
5. Lee, S., Puig-Suari, J., Twiggs, R.J., "CubeSat Design Specifications Document," Revision VIII, Aug. 2003, pp. 1-6, URL: http://cubesat.calpoly.edu.
6. Lin, Shu, *Error Control Coding: Fundamentals and Applications*, Prentice-Hall, Englewood Cliffs, NJ, 1983, Ch. 1-4.
7. Lynch, Thomas J., *Data Compression: Techniques and Applications*, Lifetime Learning Publications, Belmont, CA, 1985, Ch. 1-11.
8. Marchant, Jason P., *Lossless Compression Using Binary Necklace Classes and Multiple Huffman Tree*, Thesis, Computer Science Department, Cal Poly, San Luis Obispo, 2001.
9. Miyahira, Tetsuo and Gary Swift, "Evaluation of Radiation Effects in Flash Memories", MTG: MAPLD Conference, Greenbelt, MD, U.S.A., 1998, URL: http://trs-new.jpl.nasa.gov/dspace/handle/2014/20481.
10. Peterson, W. Wesley, *Error-Correcting Codes*, MIT Press, Cambridge, MA, 1972, Ch. 1-8.
11. Puig-Suari, J., Turner, C., Ahlgren, W., "Development of the Standard CubeSat Deployer and a CubeSat Class PicoSatellite," IEEE, 1 0-7803-6599-2, 2001, URL: http://cubesat.calpoly.edu/reference/cubesat_paper.pdf.
12. Rao, K.R. and P.C. Yip, ed., *The Transform and Data Compression Handbook*, CRC Press, Boca Raton, FL, 2001, Ch. 4-8.
13. Salomon, D., *A Guide to Data Compression Methods*, Springer, New York, 2002, Ch. 1-4.
14. Salomon, D., *Data Compression: The Complete Reference*, Springer, New York, 1998, Ch. 1-4.
15. Sayood, Khalid, *Introduction to Data Compression*, Morgan Kaufmann Publishers, San Francisco, 2000, Ch. 1-12.
16. Shi, Yun Q., *Image and Video Compression for Multimedia Engineering: Fundamentals, Algorithms, and Standards*, CRC Press, Boca Raton, FL, 2000, Ch. 1-9.
17. Toorian, A., Blundell, E., Puig-Suari, J., "CubeSats as Responsive Satellites," Proceedings of the 3rd Annual Responsive Space Conference, Los Angles, CA, April 2005.
18. Wikipedia Bit Plane article, URL: http://en.wikipedia.org/wiki/Bit_plane.
19. Wikipedia List of CubeSats article, URL: http://en.wikipedia.org/wiki/List_of_CubeSats.
20. Wikipedia Wavelet article, URL: http://en.wikipedia.org/wiki/Wavelet.

LIST OF DEFINITIONS

ADC(S)	Attitude Determination and Control (System)
AMSAT	The Radio Amateur Satellite Corporation
Bus	Part of the satellite that handles overall satellite functionality and communications (as opposed to the payload)
CCD	Charge-Coupled Device
CDH (C&DH)	Command and Data Handling
CMOS	Complementary Metal-Oxide-Semiconductor
CP-1	Cal Poly's first satellite
CP-2	Cal Poly's second satellite
CP-3	Cal Poly's third satellite
CP-X	The series of CubeSat spacecraft developed by PolySat
CRC	Cyclic Redundancy Check
DCT	Discrete Cosine Transform
DMA	Direct Memory Access
DSP	Digital Signal Processor
EEPROM	Electrically Erasable Programmable Read-Only Memory
EMI	Electromagnetic Interference
FAT16	File Allocation Table 16
FCC	Federal Communications Commission
GCC	GNU Compiler Collection
GIF	Graphics Interchange Format
I2C	Inter-Integrated Circuit (data bus)
ICE	In-Circuit Emulator
IDE	Integrated Development Environment
JPEG	Joint Photographic Experts Group (compression method)
JPEG2000	Joint Photographic Experts Group 2000 (compression method)
JTAG	Joint Test Action Group (debugging interface)
LSB	Least Significant Bit
LZW	Lempel-Ziv-Welch (algorithm)
MiniSD	Miniature Secure Digital (card)
MMC	MultiMediaCard
MSB	Most Significant Bit
Payload	Main experiment or function of the satellite
PDF	Portable Document Format
PIC	Programmable Intelligent Computer (microcontroller)
P-POD	Poly Picosatellite Orbital Deployer
RLE	Run-Length Encoding
SD	Secure Digital (card)
SNR	Signal-to-Noise Ratio
SPI	Serial Peripheral Interface (data bus)
SRAM	Static Random Access Memory
TIFF	

UART	Universal Asynchronous Receiver/Transmitter
VDK	Visual DSP++ Kernel
VGA	Video Graphics Array

Appendix A: Hardware Specifications

A.1 Processor

The CP-3 satellite used an Analog Devices Blackfin BF533 digital signal processor as the main payload processor. Datasheets and information can be found at:

http://www.analog.com/en/embedded-processing-dsp/blackfin/adsp-bf533/processors/product.html.

A.2 Processor Board

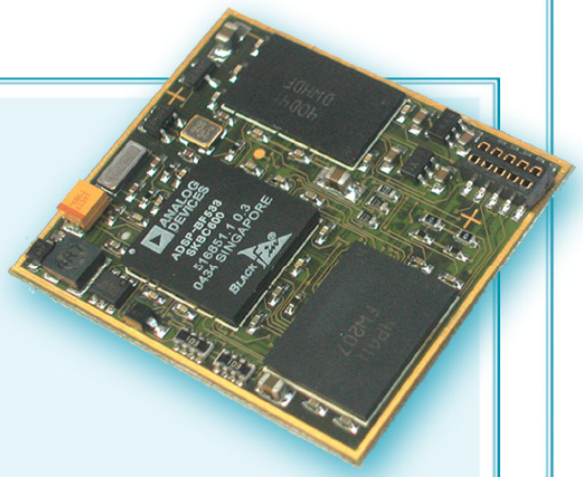
The payload purchased BlueTechnix' CM-BF533 Tinyboard module, which included the BF533, as well as associated hardware peripherals, such as memory. Datasheets and information can be found at:

http://www.bluetechnix.at/rainbow2006/site/hardware/core_modules/__cm-bf533/310/cm-bf533.aspx.

A short spec for the CM-BF533 is shown in Figure A-1 below.

CM-BF533 Core Module

The CM-BF533 is a high performance and low power processor Core Module incorporating Analog Devices Blackfin family of processors. It is the lowest cost member of the Blackfin family and can be used as a very powerful co-processor module or in a generic standalone application. The module allows easy integration into high demanding very space and power limited applications.



Features

Dimension: only 31x36mm
Blackfin BF533 @ 600MHz
32MB SD-RAM up to 133MHz
2MB Flash
25MHz Crystal
1x UART, 1x SPI,
2x SSP (Synchronous Serial Port)
1x PPI (Parallel Port Interface)
Single 3.3V supply voltage
2x 60 Pin connector 0.6mm
BLACKSheep boot loader

Development Support

EVAL-BF5xx Evaluation Board
DEV-BF5xxDA-Lite Development Board
DEV-BF5xx-FPGA Development Board
BLACKSheep VDK Framework
uClinux port available

Typical Applications

Generic Signal Processor Module
Audio and Video Applications
Prototyping, Small and Medium Volume Applications
Stand-alone Vision System

BLUETECHNIX

Waidhausenstrasse 3/19
A-1140 Wien
AUSTRIA

Sales +43 (1) 914 20 91 x 0
Web www.bluetechnix.com
Email office@bluetechnix.at

Figure A-1: CM-BF533 Short Spec

A.3 Imagers

The CP-3 satellite used two imagers, a grayscale Kodak KAC-9638, and its color cousin, the KAC-9648. Datasheets and information can be found at:

http://www.kodak.com/global/en/business/ISS/Products/CMOS/KAC-9638/overview.jhtml?pq-path=11978 and

http://www.kodak.com/global/en/business/ISS/Products/CMOS/KAC-9648/overview.jhtml?pq-path=11937/11938/11939/11980.

Short specs for the KAC-9638 and KAC-9648 are shown in Figures A-2 and A-3 below.

Kodak
Image Sensor Solutions

PRODUCT SUMMARY

KODAK KAC-9638 IMAGE SENSOR

1288 (H) X 1032 (V) MONOCHROME CMOS IMAGE SENSOR

DESCRIPTION

The KODAK KAC-9638 Image Sensor is a high performance, low power, 1/2" SXGA CMOS Active Pixel Sensor capable of capturing still, or motion images and converting them to a digital data stream.

Mega-pixel class image quality is achieved by integrating a high performance analog signal processor comprising of a high speed 10 bit A/D converter, fixed pattern noise elimination circuits and a programmable gain amplifier. The offset and black level can be automatically adjusted on chip using a full loop black level compensation circuit.

Furthermore, a programmable smart timing and control circuit allows the user maximum flexibility in adjusting integration time, active window size, gain, frame rate. Various control, timing and power modes are also provided.

Parameter	Typical Value
Array Format	Total: 1032 x 1312 Active: 1032 (V) x 1288 (H)
Effective Image Area	Total: 6.192mm x 7.872mm Active: 6.192mm x 7.728mm
Optical Format	1/2"
Pixel Size	6.0 µm x 6.0 µm
Video Outputs	8 & 10 Bit Digital
Frame Rate	18 frames per second
Dynamic Range	55 dB
Shutter	Rolling reset
FPN	0.2%
PRNU	1.7%
Sensitivity	2.40 V/lux*s
Fill Factor	49%
Microlens	none
Package	48 LCC
Single Supply	3.0V ± 10%
Power Consumption	150mW
Operating Temp	10° C to 50° C

FEATURES

- Video and snapshot operation
- Progressive scan read out with horizontal and vertical flip
- Programmable exposure with master clock divider, inter row delay, inter frame delay, and partial frame integration
- Programmable gain amplifier
- Full automatic servo loop for black level & offset adjustment on each gain channel
- Horizontal & vertical sub-sampling (2:1 & 4:2) with averaging
- Windowing
- Programmable pixel clock, inter-frame and inter-line delays
- I^2C compatible serial control interface

APPLICATIONS

- Security Camera
- Machine Vision
- Barcode Scanners
- Biometrics

Figure A-1: KAC-9638 Short Spec

Kodak
Image Sensor Solutions

PRODUCT SUMMARY

KODAK KAC-9648 IMAGE SENSOR

1288 (H) X 1032 (V) COLOR CMOS IMAGE SENSOR

DESCRIPTION

The KODAK KAC-9648 Image Sensor is a high performance, low power, 1/2" SXGA CMOS Active Pixel Sensor capable of capturing color, monochrome, still, or motion images and converting them to a digital data stream.

Mega-pixel class image quality is achieved by integrating a high performance analog signal processor comprising of a high speed 10 bit A/D converter, fixed pattern noise elimination circuits and separate color gain amplifiers. The offset and black level can be automatically adjusted on chip using a full loop black level compensation circuit.

Furthermore, a programmable smart timing and control circuit allows the user maximum flexibility in adjusting integration time, active window size, gain, frame rate. Various control, timing and power modes are also provided.

Parameter	Typical Value
Array Format	Total: 1032 x 1312 Active: 1032 (V) x 1288 (H)
Effective Image Area	Total: 6.192mm x 7.872mm Active: 6.192mm x 7.728mm
Optical Format	1/2"
Pixel Size	6.0 µm x 6.0 µm
Video Outputs	8 & 10 Bit Digital
Frame Rate	18 frames per second
Dynamic Range	55 dB
Shutter	Rolling reset
FPN	0.2%
PRNU	1.7%
Sensitivity	2.5 V/lux*s
Fill Factor	49%
Color Mosaic	Bayer pattern
Microlens	none
Package	48 LCC
Single Supply	3.0V ± 10%
Power Consumption	150mW
Operating Temp	-10° C to 50° C

FEATURES

- Video and snapshot operation
- Progressive scan read out with horizontal and vertical flip
- Programmable exposure with master clock divider, inter row delay, inter frame delay, and partial frame integration
- Four channels of digitally programmable analog gain
- Full automatic servo loop for black level & offset adjustment on each gain channel
- Horizontal & vertical sub-sampling (2:1 & 4:2) with averaging
- Windowing
- Programmable pixel clock, inter-frame and inter-line delays
- I^2C compatible serial control interface

APPLICATIONS

- Dual Mode Camera
- Digital Still Camera
- Security Camera
- Machine Vision

Figure A-2: KAC-9648 Short Spec

Appendix B: Bus and Payload Schematics

An overview of PolySat's generic satellite bus is shown in Figure B-1 below. An overview of CP-3's payload connections is shown in Figure B-2 below. Figure B-3 shows the imager daughterboard schematic.

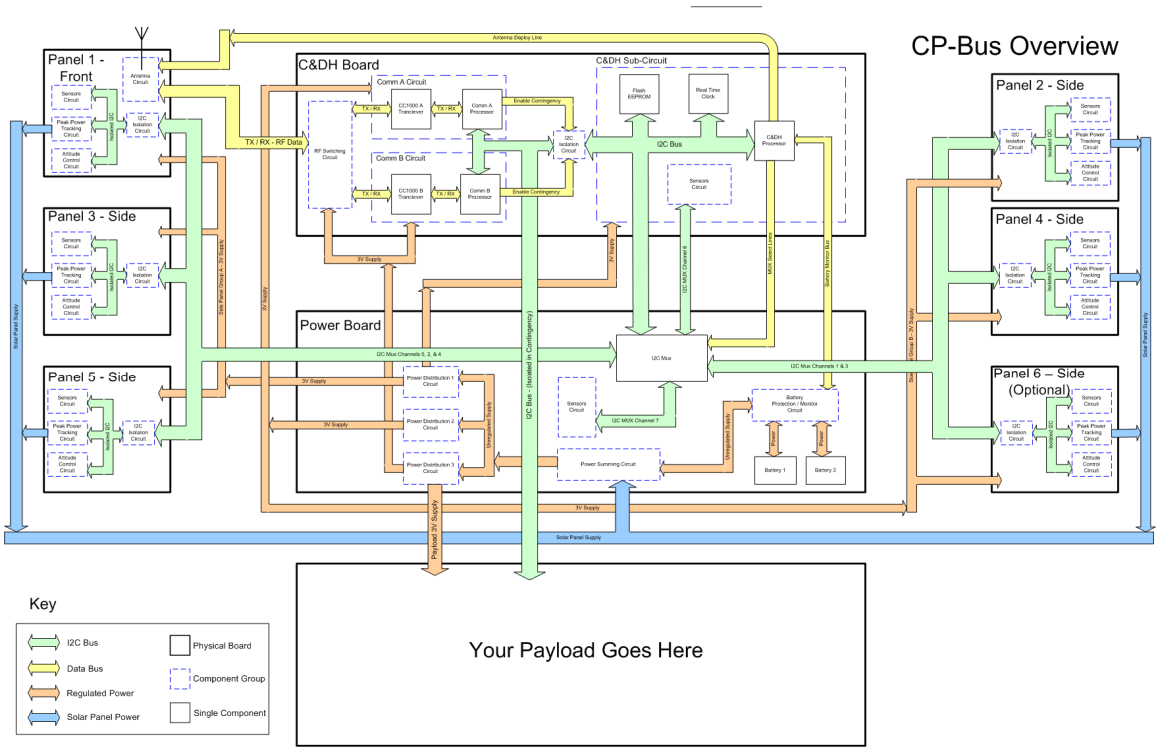


Figure B-1: Generic Satellite Bus

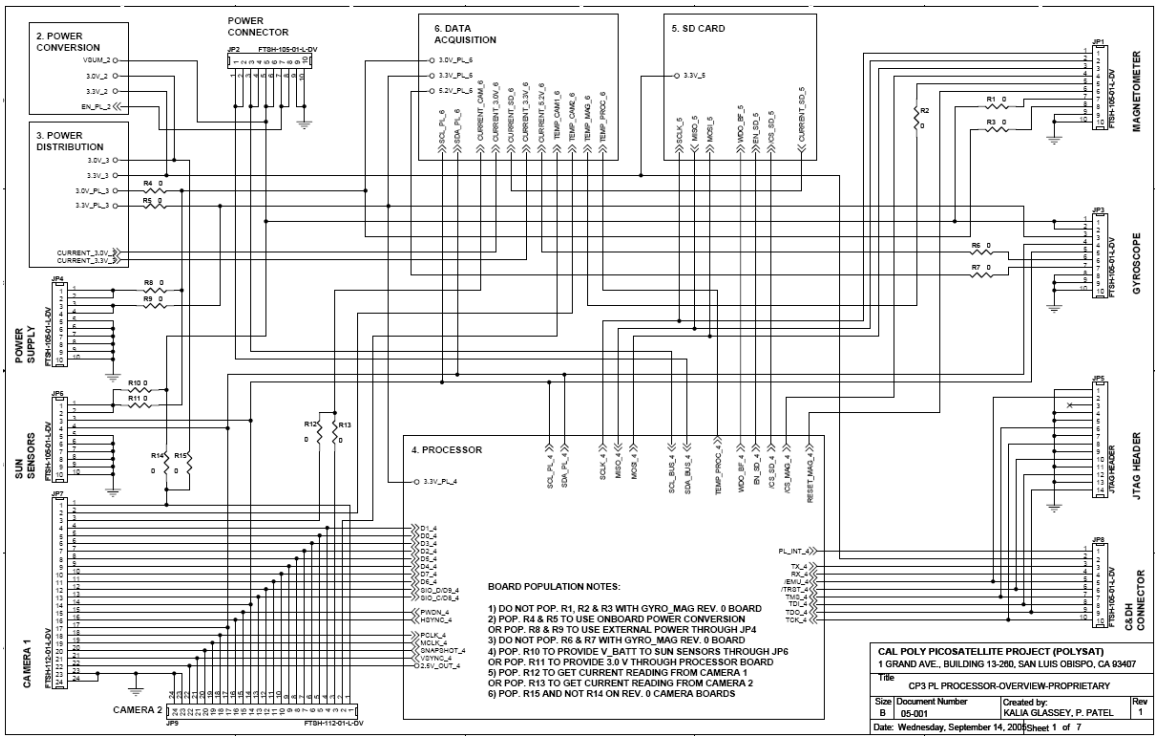


Figure B-2: Payload Overview

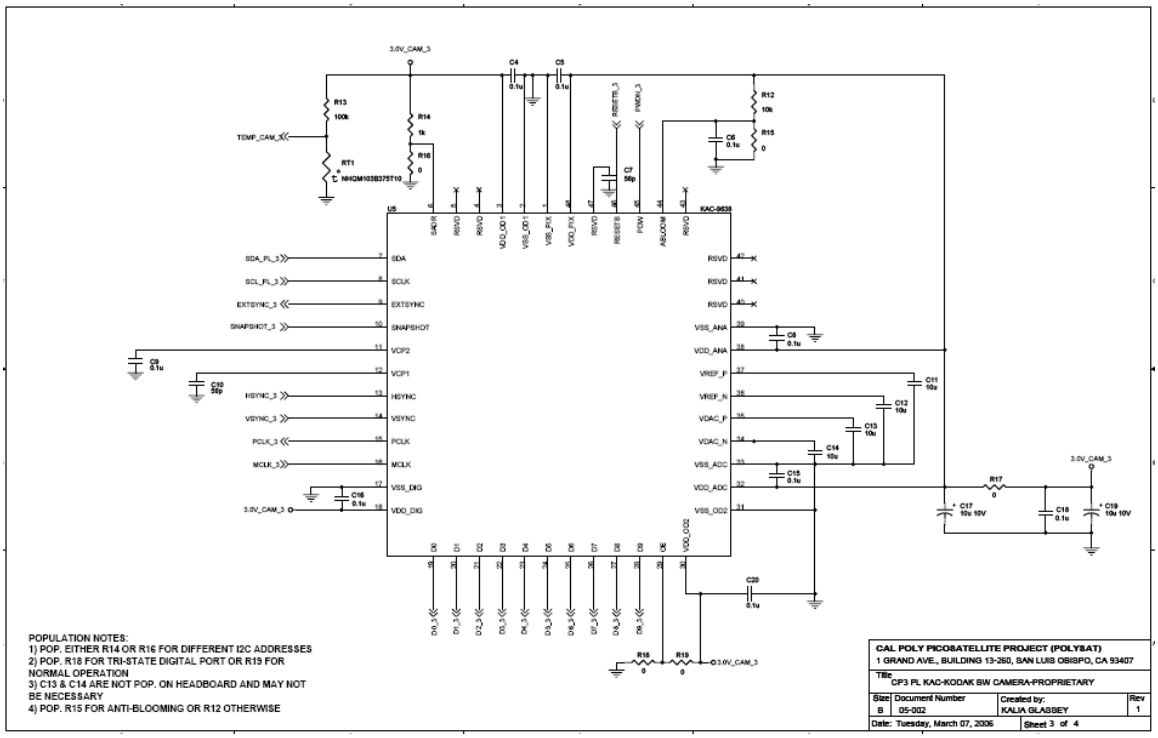


Figure B-3: Imager Daughterboard Schematic

Appendix C: Payload Software Development

Most of the bus hardware and software was a legacy from the previous satellite, CP-2. Thus, besides minor changes to the bus software, the vast majority of the software development related to the new payload processor.

C.1 Payload Operating System

The software team based payload software development on a keep-it-simple philosophy. The payload processor used Analog Devices' VDK (Visual DSP++ Kernel) real-time operating system. While the main PIC processor on the bus used a simple while loop to coordinate processor tasks, VDK's multi-tasking system was deemed to be more flexible, albeit also more complicated. In addition to the imagers, the payload hardware included multiple sensors and other peripherals, necessitating multi-tasking on the processor's part. In addition, the operating system needed to respond to various interrupts and allow for thread pre-emption if another task assumed a higher priority.

C.2 Payload/Bus Interface

The payload/bus interface utilized the bus's main I2C data bus. While this was convenient because it did not need to be built from scratch, it had several drawbacks. The most serious drawback was that the data bus ran at 100 kHz, which made the payload/bus connection a significant choke point in transferring large amounts of data from the payload to the ground. Because the bus processor only had a 256 byte buffer, each communication had to be 256 bytes or less. In addition, the I2C bus protocol does not allow a slave to initiate communications. Since the payload was a slave to the

satellite bus, it was necessary to add a hardware interrupt line from the payload to the bus processor that let the bus know when the payload had information to communicate.

Because of concerns about radiation's effect on the flash memory (see section 3.3.2), CRC coding was used when data was transferred from the SD card to the bus.

C.3 Payload Peripherals Buses

The payload processor communicated with its peripherals in a number of ways. Many of the sensors, including the imager, had I2C control lines. In addition, an SPI bus was required for the SD card, as well as a parallel connection between the processor and the imager.

C.4 Flash Memory

Because the built-in memory of the processor was not enough to store more than a few images, more memory was clearly necessary. The payload uses a large amount of power, so it is designed to be turned off except while running experiments, making it necessary to obtain non-volatile memory. Non-volatile SRAM is hard to obtain in large-density packages, and would have required an excessive amount of volume. Thus the decision was made to use flash memory. The mini-SD card was chosen as it was one of the highest-density options available at the time. Unfortunately, it suffers from susceptibility to cosmic radiation [9]. To combat this problem, error-checking was instituted in the file system to ensure data integrity.

Rather than use a custom-designed file system, the SD card used a FAT16 file system. This was done to reduce design complexity since open-source code existed for creating the FAT16 system. In addition, during testing it was convenient because the SD

card could be directly inserted into a standard computer card reader due to file system compatibility.

C.5 Imager Software

Taking a picture consists of the following sequence of events:

- load file with register settings from SD card into processor memory
- modify register settings using the I2C bus
- assert a "snapshot" pin on the imager to take a picture
- wait for the imager's pixel clock to begin running, indicating the image is complete
- use DMA to transfer the image directly to processor memory
- write image from processor memory to SD card over SPI bus

The processor chosen by the team was designed for image processing, and so had dedicated vertical sync, horizontal sync, and frame sync pins that could be used to automatically store an image using DMA. Unfortunately, the vertical and horizontal sync pins were wired backwards in the hardware layout, meaning the imaging system couldn't take advantage of this feature. Therefore, the more complicated technique of using the imager in snapshot mode and listening for the pixel clock was required.

Appendix D: Testing Data

Thermal/vacuum testing was undertaken to ensure that component temperatures stayed within specifications, and that voltages and currents did not fluctuate excessively. The following Figures D-1 through D-3 show payload thermal/vacuum testing data.

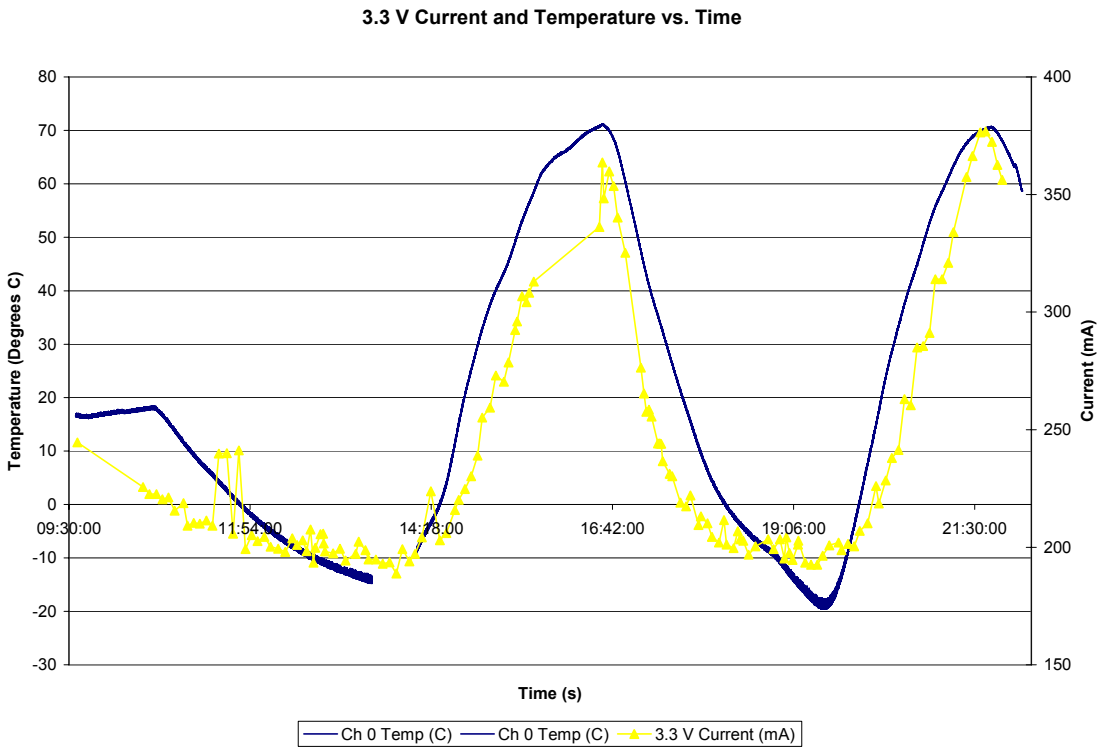


Figure D-1: 3.3 V Power Supply Current Draw with Respect to Temperature

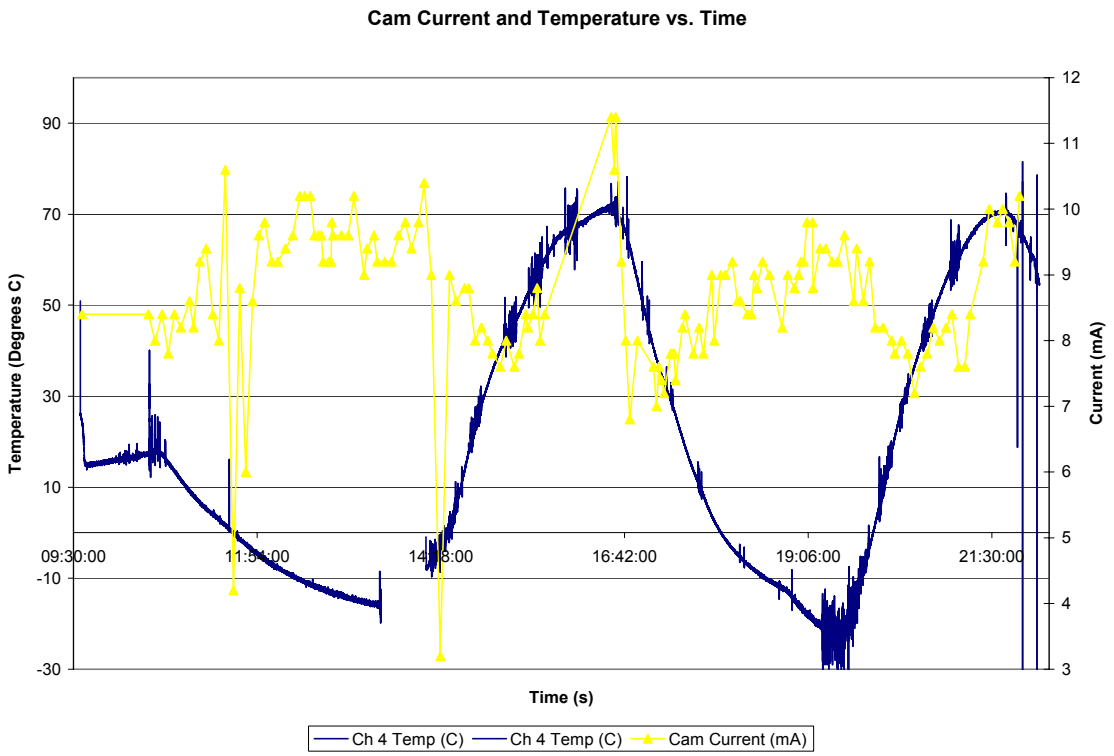


Figure D-2: Camera Quiescent Current Draw with Respect to Temperature

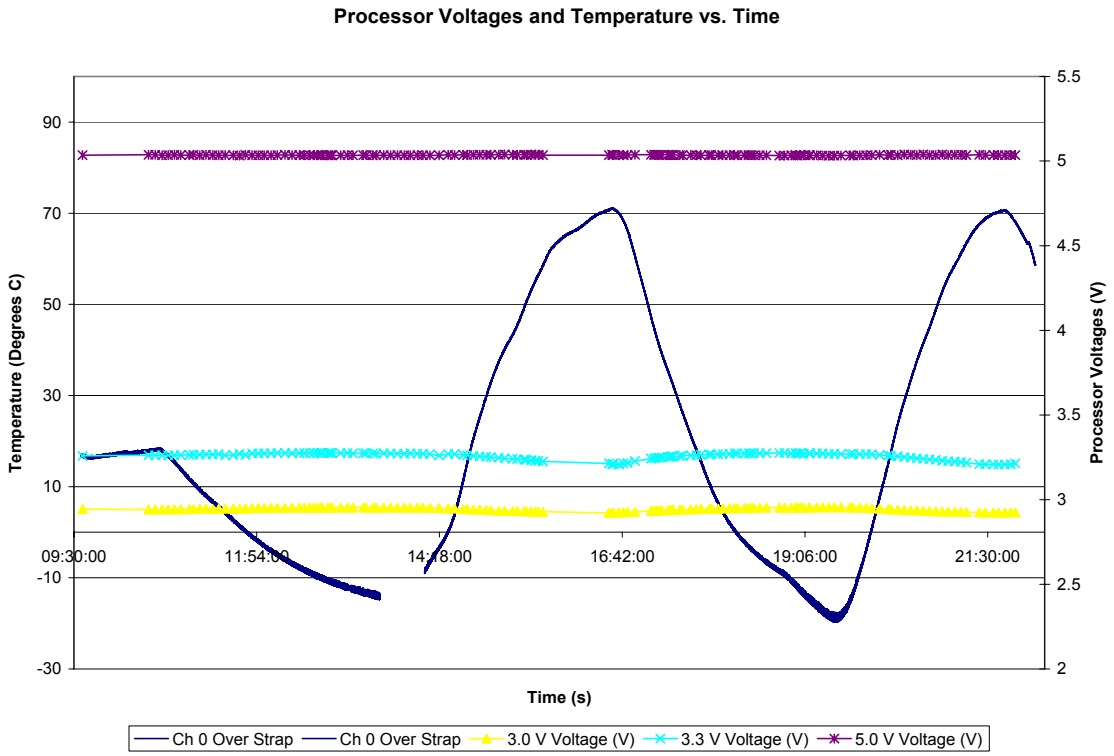


Figure D-3: Processor Voltages with Respect to Temperature

Lens testing was necessary to determine which lenses produced the least distortion while letting in as much light as possible. The following images shown in Figure D-4 show lens comparison tests taken with identical imager settings.

Figure D-4: Lens Comparison Tests

As the above images show, the different lenses gave significantly different results in terms of distortion, field-of-view, and light transmission.

Appendix E: Source Code

This appendix contains the compression program source code. The main C file is entitled ‘main.c’, with supporting C files ‘rawio.c’, ‘stats.c’, ‘huff.c’, ‘LZW.c’, ‘predictive.c’, ‘bitplane.c’, and ‘binarycomp.c’ and their associated header files.

main.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <sys/stat.h>
#include "rawio.h"
#include "stats.h"
#include "huff.h"
#include "LZW.h"
#include "predictive.h"
#include "bitplane.h"
#include "binarycomp.h"

#define NUM_FILES 1                         //number of images under test
#define NUM_TESTS 29                        //number of tests per image

//image names and numbers of rows/columns
#define IMAGE_0 "sample.raw"
#define NUM_ROWS_0 512
#define NUM_COLS_0 512

int main(void)
{
   ////////////////////////////////////////////////////////////////////
   //Variable Declaration

   int i,j,m;                               //indexing integers
   int *array;                              //for predictive coding test
   unsigned int num_rows[NUM_FILES], num_cols[NUM_FILES];
   unsigned int entries;                    //for LZW test
   unsigned long int *samplehist;   //histogram for compression tests
   char *image_name[NUM_FILES];
   char test_result;                //result of comparison test (0 for
                                    //different, 1 for same)
   char *r_name = "results.txt";            //results output file name
   struct stat file_status;                 //file variable
   FILE *stream;                            //for I/O function
   RAW_IMG *sample, *test, *bpesample, *bpimg[8], *testbpimg[8];
   HUFF_DICT *huffman;                      //Huffman dictionary
   BIT_PLANE *samplebp, *testbp;
   RESULT *results[NUM_FILES][NUM_TESTS]; //results array

   ////////////////////////////////////////////////////////////////////
   //Test Setup and Memory Allocation

   //allocate memory
   for (m = 0; m < 8; m++)
   {
      bpimg[m] = malloc(sizeof(RAW_IMG));
      testbpimg[m] = malloc(sizeof(RAW_IMG));
   }

   samplebp = malloc(sizeof(BIT_PLANE));
   testbp = malloc(sizeof(BIT_PLANE));
```

```
for (i = 0; i < NUM_FILES; i++)
{
   for (j = 0; j < NUM_TESTS; j++)
   {
      results[i][j] = malloc(sizeof(RESULT));
      results[i][j]->image_name = malloc(30*sizeof(char));
      results[i][j]->test_name = malloc(30*sizeof(char));
   }
   image_name[i] = malloc(30*sizeof(char));
}

//assign image characteristics
image_name[0] = &IMAGE_0;
num_rows[0] = NUM_ROWS_0;
num_cols[0] = NUM_COLS_0;


////////////////////////////////////////////////////////////////////
//Compression tests

for (i = 0; i < NUM_FILES; i++)
{
   printf("Running compression tests on image %d of
                                                %d\n",i+1,NUM_FILES);

   j = 0;            //initialize counter

   //read raw image
   sample = read_raw(image_name[i],num_rows[i], num_cols[i]);
   sample->code = 0;       //binary coded image


   ////////////////////////////////////////////////////////////////
   //Huffman Encoding

   printf("Huffman Encoding...\n");

   //encode image and write compressed file
   sample->raw_image_name = "Huffman.cmp";
   samplehist = hist(sample);
   huffman = hdcreate(samplehist,256);
   huffenco(huffman,sample);

   //decode image and write decompressed image
   test = huffdeco("Huffman.cmp",num_rows[i],num_cols[i]);
   test->raw_image_name = "Huffman.dec";
   write_raw(test,test->raw_image_name);

   //compare decoded image to original
   test_result = compareimg(sample, test);

   //record results
   results[i][j]->test_name = "Huffman";
   results[i][j]->image_name = image_name[i];
   results[i][j]->result = test_result;
   //calculate size of compressed file
   if (stat(sample->raw_image_name, &file_status) != 0)
```

```
   perror("Could not determine compressed file size\n");
results[i][j]->comp_size = file_status.st_size;
//calculate size of original file
if (stat(image_name[i], &file_status) != 0)
   perror("Could not determine original file size\n");
results[i][j]->image_size = file_status.st_size;
results[i][j]->num_rows = sample->num_rows;
results[i][j]->num_cols = sample->num_cols;

//delete output files
remove(sample->raw_image_name);
remove(test->raw_image_name);

j++;  //increment j


////////////////////////////////////////////////////////////////
//LZW Encoding

printf("LZW Encoding...\n");

//encode image and write compressed file
sample->raw_image_name = "LZW.cmp";
samplehist = hist(sample);
entries = LZWenco(sample->raw_image_name,sample,samplehist);

//decode image and write decompressed image
test = LZWdeco(sample-
     >raw_image_name,samplehist,num_rows[i],num_cols[i],entries);
test->raw_image_name = "LZW.dec";
write_raw(test,test->raw_image_name);

//compare decoded image to original
test_result = compareimg(sample, test);

//record results
results[i][j]->test_name = "LZW";
results[i][j]->image_name = image_name[i];
results[i][j]->result = test_result;
//calculate size of compressed file
if (stat(sample->raw_image_name, &file_status) != 0)
   perror("Could not determine compressed file size\n");
results[i][j]->comp_size = file_status.st_size;
//calculate size of original file
if (stat(image_name[i], &file_status) != 0)
   perror("Could not determine original file size\n");
results[i][j]->image_size = file_status.st_size;
results[i][j]->num_rows = sample->num_rows;
results[i][j]->num_cols = sample->num_cols;

//delete output files
remove(sample->raw_image_name);
remove(test->raw_image_name);

j++;  //increment j
```

```
////////////////////////////////////////////////////////////////
//Predictive Encoding

//set up test array and create histogram
array = predenco(sample,1);

samplehist = hista(array,num_rows[i]*num_cols[i]);

//Predictive + Huffman encoding

printf("Predictive Huffman Encoding...\n");

//encode image and write compressed files
sample->raw_image_name = "PredHuff.cmp";
huffman = hdcreate(samplehist,511);

huffencoa(huffman,array,num_rows[i]*num_cols[i],sample-
                                                >raw_image_name);

//decode Huffman image and write decompressed image
array = huffdecoa(sample-
                        >raw_image_name,num_rows[i]*num_cols[i]);
test = preddeco(array,1,sample);
test->raw_image_name = "PredHuff.dec";
write_raw(test,test->raw_image_name);

//compare decoded Huffman image to original
test_result = compareimg(sample, test);

//record results
results[i][j]->test_name = "Predictive, Huffman";
results[i][j]->image_name = image_name[i];
results[i][j]->result = test_result;
//calculate size of compressed file
if (stat(sample->raw_image_name, &file_status) != 0)
   perror("Could not determine compressed file size\n");
results[i][j]->comp_size = file_status.st_size;
//calculate size of original file
if (stat(image_name[i], &file_status) != 0)
   perror("Could not determine original file size\n");
results[i][j]->image_size = file_status.st_size;
results[i][j]->num_rows = sample->num_rows;
results[i][j]->num_cols = sample->num_cols;

//delete output files
remove(sample->raw_image_name);
remove(test->raw_image_name);

j++;  //increment j

//Predictive + LZW encoding

printf("Predictive LZW Encoding...\n");

//encode image and write compressed files
sample->raw_image_name = "PredLZW.cmp";
```

```
entries = LZWencoa(sample-
        >raw_image_name,array,num_rows[i]*num_cols[i],samplehist);

//decode LZW image and write decompressed image
array = LZWdecoa(sample-
     >raw_image_name,samplehist,num_rows[i]*num_cols[i],entries);
test = preddeco(array,1,test);
test->raw_image_name = "PredLZW.dec";
write_raw(test,test->raw_image_name);

//compare decoded LZW image to original
test_result = compareimg(sample, test);

//record results
results[i][j]->test_name = "Predictive, LZW";
results[i][j]->image_name = image_name[i];
results[i][j]->result = test_result;
//calculate size of compressed file
if (stat(sample->raw_image_name, &file_status) != 0)
   perror("Could not determine compressed file size\n");
results[i][j]->comp_size = file_status.st_size;
//calculate size of original file
if (stat(image_name[i], &file_status) != 0)
   perror("Could not determine original file size\n");
results[i][j]->image_size = file_status.st_size;
results[i][j]->num_rows = sample->num_rows;
results[i][j]->num_cols = sample->num_cols;

//delete output files
remove(sample->raw_image_name);
remove(test->raw_image_name);

j++;  //increment j


////////////////////////////////////////////////////////////////
//Bit Plane Encoding
//    Divides image into bit planes, and encodes and decodes them
//using various algorithms and reports results
//  Bit planes are generated both from original image and Gray
//coded image

printf("Bitplane Encoding...\n\n");

bpesample = sample;

//slice image into bit planes
samplebp = sliceimage(bpesample);

for (m = 0; m < 8; m++)
{
   bpimg[m] = bptoimg(samplebp->bplane[m]);     //turn lower 7
                                        //bit planes to RAW_IMGs
}

for (m = 0; m < 8; m++)
{
```

```
//Huffman

//encode image and write compressed file
bpimg[m]->raw_image_name = "BPHuff.cmp";
samplehist = hist(bpimg[m]);
huffman = hdcreate(samplehist,256);
huffenco(huffman,bpimg[m]);

//decode image and write decompressed image
testbpimg[m] = huffdeco(bpimg[m]->raw_image_name,bpimg[m]-
                                >num_rows,bpimg[m]->num_cols);
testbpimg[m]->raw_image_name = "BPHuff.dec";
write_raw(testbpimg[m],testbpimg[m]->raw_image_name);

//compare decoded images to originals
test_result = compareimg(bpimg[m], testbpimg[m]);

//record results
results[i][j]->test_name = "BP, Huff, Single";
results[i][j]->image_name = image_name[i];
results[i][j]->result = test_result;
//calculate size of compressed file
if (stat(bpimg[m]->raw_image_name, &file_status) != 0)
   perror("Could not determine compressed file size\n");
results[i][j]->comp_size = file_status.st_size;
//calculate size of original file
if (stat(testbpimg[m]->raw_image_name, &file_status) != 0)
   perror("Could not determine original file size\n");
results[i][j]->image_size = file_status.st_size;
results[i][j]->num_rows = bpimg[m]->num_rows;
results[i][j]->num_cols = bpimg[m]->num_cols;

//delete output files
remove(bpimg[m]->raw_image_name);
remove(testbpimg[m]->raw_image_name);

j++;     //increment j

//LZW

//encode image and write compressed file
bpimg[m]->raw_image_name = "BPLZW.cmp";
entries = LZWenco(bpimg[m]->raw_image_name, bpimg[m],
                                                  samplehist);
               //LZW encode; histogram already done above

//decode image and write decompressed image
testbpimg[m] = LZWdeco(bpimg[m]->raw_image_name, samplehist,
               bpimg[m]->num_rows,bpimg[m]->num_cols,entries);
               //LZW decode
testbpimg[m]->raw_image_name = "BPLZW.dec";
write_raw(testbpimg[m],testbpimg[m]->raw_image_name);

//compare decoded images to originals
test_result = compareimg(bpimg[m], testbpimg[m]);

//record results
```

```
results[i][j]->test_name = "BP, LZW, Single";
results[i][j]->image_name = image_name[i];
results[i][j]->result = test_result;
//calculate size of compressed file
if (stat(bpimg[m]->raw_image_name, &file_status) != 0)
   perror("Could not determine compressed file size\n");
results[i][j]->comp_size = file_status.st_size;
//calculate size of original file
if (stat(testbpimg[m]->raw_image_name, &file_status) != 0)
   perror("Could not determine original file size\n");
results[i][j]->image_size = file_status.st_size;
results[i][j]->num_rows = bpimg[m]->num_rows;
results[i][j]->num_cols = bpimg[m]->num_cols;

//delete output files
remove(bpimg[m]->raw_image_name);
remove(testbpimg[m]->raw_image_name);

j++;     //increment j

//Zero RLC encoding

//allocate memory
testbp->bplane[m] = malloc(sizeof(PLANE));
if (!(testbp->bplane[m]))
   printf("Insufficient memory available\n");
testbp->bplane[m]->num_bytes = samplebp->bplane[m]->num_bytes;
testbp->bplane[m]->comptype = 0; //no compression by default
testbp->bplane[m]->bp = calloc(testbp->bplane[m]-
                                                >num_bytes,1);

//encode image
testbp->bplane[m] = nzcode(samplebp->bplane[m]);

//record size of compressed bit plane
results[i][j]->comp_size = testbp->bplane[m]->num_bytes;

//decode image
testbp->bplane[m] = nzdecode(testbp->bplane[m],samplebp-
                                        >bplane[m]->num_bytes);

//format bit planes as images
testbpimg[m] = bptoimg(testbp->bplane[m]);

//compare decoded images to originals
test_result = compareimg(bpimg[m], testbpimg[m]);

//record results
results[i][j]->test_name = "BP, NZ, Single";
results[i][j]->image_name = image_name[i];
results[i][j]->result = test_result;
//record size of original image
results[i][j]->image_size = bpimg[m]->num_rows*bpimg[m]-
                                                    >num_cols;
results[i][j]->num_rows = bpimg[m]->num_rows;
results[i][j]->num_cols = bpimg[m]->num_cols;
```

```
            j++;     //increment j

         }

         //rejoin image
         test = joinimage(samplebp);

         //compare decoded images to originals
         test_result = compareimg(sample, test);

         //record results
         results[i][j]->test_name = "BP, Huff, Complete";
         results[i][j]->image_name = image_name[i];
         results[i][j]->result = test_result;
         //no complete compressed file
         results[i][j]->comp_size = 0;
         //calculate size of original file
         if (stat(image_name[i], &file_status) != 0)
            perror("Could not determine original file size\n");
         results[i][j]->image_size = file_status.st_size;
         results[i][j]->num_rows = bpesample->num_rows;
         results[i][j]->num_cols = bpesample->num_cols;

         j++;  //increment j
      }

      ////////////////////////////////////////////////////////////////////
      //Report results

      stream = fopen(r_name,"wt");
      if (stream==NULL)
         printf("The file was not opened\n");

      fprintf(stream,"Image Name:\t\tTest Name:\t\t\tTest
         Result:\tCompressed Size:\tOriginal Size:\tRows:\tColumns:\n\n");
      for (i = 0; i < NUM_FILES; i++)
      {
         for (j = 0; j < NUM_TESTS; j++)
         {
            fprintf(stream,"%s\t",results[i][j]->image_name);
            fprintf(stream,"%-25s\t",results[i][j]->test_name);
            if (results[i][j]->result == 0)
               fprintf(stream,"No Match\t");
            else if (results[i][j]->result == 1)
               fprintf(stream,"Match\t\t");
            fprintf(stream,"%16d\t",results[i][j]->comp_size);
            fprintf(stream,"%14d\t",results[i][j]->image_size);
            fprintf(stream,"%5d\t",results[i][j]->num_rows);
            fprintf(stream,"%8d\n",results[i][j]->num_cols);
         }
      }
      fclose(stream);

      return 0;
   }
```

rawio.h:

```
#ifndef INCLUDE_RAWIO
#define INCLUDE_RAWIO

#include <stdio.h>

typedef struct
{
   unsigned char code;              // 0 for binary, 1 for Gray code
   int num_rows;
   int num_cols;
   char *raw_image_name;
   unsigned char *raw_image;
}
RAW_IMG;

typedef struct
{
   unsigned char bytebuffer;
   unsigned char buff_bits;
}
BYTE_BUFFER;

//reads in raw image
RAW_IMG *read_raw(char *file_name, int num_rows, int num_cols);

//writes out raw image
void write_raw(RAW_IMG *img, char *file_name);

//opens file to write out to using write_bits
//initializes BYTE_BUFFER object
//takes file to be written to as input
//returns FILE object
FILE *openwritebuffer(char *filename, BYTE_BUFFER *tempbuffer);

//writes out the number of least significant bits specified of an array
//must call open_buffer function first
//after call, bytebuffer and buff_bits point to addresses of
// bytebuffer, and the number of bits it contains
//returns number of entries written
unsigned long int write_bits(FILE *stream, BYTE_BUFFER *tempbuffer,
unsigned long int *codearray, unsigned long int array_entries, unsigned
char num_bits);

//flushes byte buffer and returns number of bits flushed
//pads extra bits with 0's
//must call after finished writing to file
unsigned char flushwritebuffer(FILE *stream, BYTE_BUFFER *tempbuffer);

//opens file to read in from using read_bits
//takes file to be read from as input
//initializes BYTE_BUFFER object
//returns FILE object
FILE *openreadbuffer(char *filename, BYTE_BUFFER *readbuffer);
```

```
//reads streamed file into codearray (max_entries possible values) and
// returns number of values read
//num_bits indicates how many bits each entry is
//must close stream manually
unsigned long int read_bits(FILE *stream, BYTE_BUFFER *readbuffer,
unsigned long int *codearray, unsigned long int max_entries, unsigned
char num_bits);

//convert binary raw image to Gray coded image
RAW_IMG *bintogcode(RAW_IMG *raw_image);

//convert Gray coded raw image to binary
RAW_IMG *gcodetobin(RAW_IMG *raw_image);

#endif
```

rawio.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "rawio.h"
#include "bitplane.h"

//reads in raw image
RAW_IMG *read_raw(char *file_name, int num_rows, int num_cols)
{
   long int numread;
   FILE *stream;

   //allocate memory for storing image
   RAW_IMG *img = malloc(sizeof(RAW_IMG));
   if (img == NULL)
      printf( "Insufficient memory available\n" );

   img->raw_image = calloc(num_rows*num_cols,1);
   if(img->raw_image == NULL)
   {
      printf( "Insufficient memory available\n" );
   }

   //open image file
   stream = fopen(file_name,"rb");
   if(stream == NULL)
      printf( "The file was not opened\n" );

   //read image file into memory
   numread = fread(img->raw_image,1,num_rows*num_cols,stream);

   if (numread < num_rows*num_cols)
      printf("Not all data read\n");

   //close file
   fclose(stream);

   //set RAW_IMG characteristics
   img->num_cols = num_cols;
   img->num_rows = num_rows;
   img->raw_image_name = file_name;
   img->code = 0;       //assume binary code
   return img;
}

//writes out raw image
void write_raw(RAW_IMG *img, char *file_name)
{
   int numwritten;
   FILE *stream;

   //open file for writing image
   stream = fopen(file_name,"wb");
   if (stream==NULL)
```

```
      printf("The file was not opened\n");

   //write image to file
   numwritten = fwrite(img->raw_image,1,(img->num_cols)*(img-
                                                >num_rows),stream);
   if (numwritten < (img->num_cols)*(img->num_rows))
      printf("Not all data written\n");

   fclose(stream);
}

//opens file to write out to using write_bits
//initializes BYTE_BUFFER object
//takes file to be written to as input
//returns FILE object
FILE *openwritebuffer(char *filename, BYTE_BUFFER *tempbuffer)
{
   FILE *stream;
   stream = fopen(filename,"wb");
   if (stream == NULL)
      printf("The file was not opened\n");
   tempbuffer->bytebuffer = 0;
   tempbuffer->buff_bits = 0;
   return stream;
}

//writes out the number of least significant bits specified of an array
//must call open_buffer function first
//after call, bytebuffer and buff_bits point to addresses of
// bytebuffer, and the number of bits it contains
//returns number of entries written
unsigned long int write_bits(FILE *stream, BYTE_BUFFER *tempbuffer,
                        unsigned long int *codearray, unsigned long int
                        array_entries, unsigned char num_bits)
{
   unsigned char bit_num, bit, shift, numwritten;
   unsigned long int mask, i;

   for (i = 0; i < array_entries; i++)
   {
      for (bit_num = 1; bit_num <= num_bits; bit_num++)
      {
         //shift bits into buffer
         shift = num_bits-bit_num;
         mask = 1 << (shift);
         bit = (unsigned char)((codearray[i] & mask)>>shift);
         tempbuffer->bytebuffer = (tempbuffer->bytebuffer << 1) + bit;
         tempbuffer->buff_bits++;

         //if buffer is full, write one byte to file
         if (tempbuffer->buff_bits == 8)
         {
            numwritten = fwrite(&(tempbuffer->bytebuffer),1,1,stream);
            if (numwritten != 1)
               printf("Data write error\n");
            tempbuffer->bytebuffer = 0;
            tempbuffer->buff_bits = 0;
```

```
            }
        }
    }
    return i;
}

//flushes byte buffer and returns number of bits flushed
//pads extra bits with 0's
//must call after finished writing to file
unsigned char flushwritebuffer(FILE *stream, BYTE_BUFFER *tempbuffer)
{
    unsigned char temp_bits, numwritten;

    //if buffer has extra bits left over...
    if (tempbuffer->buff_bits != 0)
    {
        //...shift to most significant places, pad with zeros, and write
        // to file
        tempbuffer->bytebuffer = tempbuffer->bytebuffer << (8-tempbuffer-
                                                            >buff_bits);
        numwritten = fwrite(&(tempbuffer->bytebuffer),1,1,stream);
        if (numwritten != 1)
            printf("Data write error\n");
    }

    fclose(stream);

    //reinitialize buffer
    tempbuffer->bytebuffer = 0;
    temp_bits = tempbuffer->buff_bits;
    tempbuffer->buff_bits = 0;

    return temp_bits;
}

//opens file to read in from using read_bits
//takes file to be read from as input
//initializes BYTE_BUFFER object
//returns FILE object
FILE *openreadbuffer(char *filename, BYTE_BUFFER *readbuffer)
{
    FILE *stream;
    stream = fopen(filename,"rb");
    readbuffer->bytebuffer = 0;
    readbuffer->buff_bits = 0;
    if (stream == NULL)
        printf("File not opened for reading\n");
    return stream;
}

//reads file into codearray (max_entries possible values) and returns
// number of values read
//num_bits indicates how many bits each entry is
//must close stream manually
unsigned long int read_bits(FILE *stream, BYTE_BUFFER *readbuffer,
unsigned long int *codearray, unsigned long int max_entries, unsigned
char num_bits)
```

```
{
   unsigned long int numread, i;
   unsigned char bit_num, bit, mask, shift;

   for (i = 0; i < max_entries; i++)
   {
      codearray[i] = 0;
      for (bit_num = 1; bit_num <= num_bits; bit_num++)
      {
         if (readbuffer->buff_bits == 0)
         {
            numread = fread(&(readbuffer->bytebuffer),1,1,stream);
            if (numread != 1)
               printf("Data read error\n");
            readbuffer->buff_bits = 8;
         }
         shift = readbuffer->buff_bits-1;
         mask = 1 << (shift);
         bit = (readbuffer->bytebuffer & mask) >> shift;
         codearray[i] = (codearray[i] << 1) + bit;
         (readbuffer->buff_bits)--;
      }
   }
   return i;
}

//convert binary raw image to Gray coded image
RAW_IMG *bintogcode(RAW_IMG *raw_image)
{
   unsigned char mask, shift, bit[8], gcbit[8], k;
   unsigned long int num_pixels = (raw_image->num_cols*raw_image-
                                                        >num_rows), i;

   if (raw_image->code)
      printf("Already Gray Coded\n");
   else
   {
      for (i = 0; i < num_pixels; i++)
      {
         //decompose pixel into bits -- bit[0] is LSB, bit[7] is MSB
         for (k = 0; k < 8; k++)
         {
            shift = 7 - k;
            mask = 1 << shift;
            bit[shift] = ((raw_image->raw_image[i])&mask)>>shift;
         }

         //switch to Gray-coded bits
         gcbit[7] = bit[7];
         for (k = 0; k < 7; k++)
         {
            gcbit[k] = (bit[k])^(bit[k+1]);
         }

         //recompose into pixel and put back in image
         raw_image->raw_image[i] = 0;
         for (k = 0; k < 8; k++)
```

```
            {
               shift = 7-k;
               raw_image->raw_image[i] = (raw_image->raw_image[i] << 1) +
                                                                  bit[shift];
            }
         }
         raw_image->code = 1;
      }
      return raw_image;
}

//convert Gray coded raw image to binary
RAW_IMG *gcodetobin(RAW_IMG *raw_image)
{
      char k;
      unsigned char mask, shift, bit[8], gcbit[8];
      unsigned long int num_pixels = (raw_image->num_cols*raw_image-
                                                         >num_rows), i;

      if (!(raw_image->code))
         printf("Already Binary Coded\n");
      else
      {
         for (i = 0; i < num_pixels; i++)
         {
            //decompose pixel into bits -- bit[0] is LSB, bit[7] is MSB
            for (k = 0; k < 8; k++)
            {
               shift = 7 - k;
               mask = 1 << shift;
               gcbit[shift] = ((raw_image->raw_image[i])&mask)>>shift;
            }

            //switch to binary bits
            bit[7] = gcbit[7];
            for (k = 6; k >= 0; k--)
            {
               bit[k] = (bit[k+1])^(gcbit[k]);
            }

            //recompose into pixel and put back in image
            raw_image->raw_image[i] = 0;
            for (k = 0; k < 8; k++)
            {
               shift = 7-k;
               raw_image->raw_image[i] = (raw_image->raw_image[i] << 1) +
                                                                  bit[shift];
            }
         }
         raw_image->code = 0;
      }
      return raw_image;
}
```

stats.h:

```
#ifndef INCLUDE_STATS
#define INCLUDE_STATS

typedef struct
{
   char *test_name;                  //test name
   char *image_name;                 //image name
   char result;          //test result: 0 for no match, 1 for match
   unsigned int comp_size;           //compressed file size in bytes
   unsigned int image_size;    //original file size in bytes
   int num_rows;                     //number of rows in original image
   int num_cols;               //number of columns in original image
}
RESULT;

//generates pointer to length 256 vector with counts of pixels having
// values from 0 to 255, respectively
unsigned long int *hist(RAW_IMG *img);

//for non unsigned char inputs
//generates pointer to length 511 vector with counts of pixels having
// values from -255 to 255
//size is number of values in array
unsigned long int *hista(int *array, unsigned long int size);

//compare two images pixel by pixel to ensure decompressed images are
// the same as original images
//returns 0 if images differ, 1 if they are the same
char compareimg(RAW_IMG *orig_img, RAW_IMG *proc_img);

#endif
```

stats.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "rawio.h"
#include "stats.h"

//generates pointer to length 256 vector with counts of pixels having
// values from 0 to 255, respectively
unsigned long int *hist(RAW_IMG *img)
{
   long int i;
   unsigned long int *histogram = calloc(256,sizeof(unsigned long
                                                              int));
   if (!histogram)
      printf("Insufficient memory available\n");

   if (histogram == NULL)
      printf( "Insufficient memory available\n" );

   for (i = 0; i < (img->num_cols)*(img->num_rows); i++)
      histogram[img->raw_image[i]]++;

   return histogram;
}

//for non unsigned char inputs
//generates pointer to length 511 vector with counts of pixels having
// values from -255 to 255
//size is number of values in array
unsigned long int *hista(int *array, unsigned long int size)
{
   unsigned long int i;
   unsigned long int *histogram = calloc(511,sizeof(unsigned long
                                                              int));
   if (!histogram)
      printf("Insufficient memory available\n");

   for (i = 0; i < size; i++)
      histogram[255 + array[i]]++;

   return histogram;
}

//compare two images pixel by pixel to ensure decompressed images are
// the same as original images
//returns 0 if images differ, 1 if they are the same
char compareimg(RAW_IMG *orig_img, RAW_IMG *proc_img)
{
   unsigned int i;
   unsigned long int num_pixels = (orig_img->num_cols)*(orig_img-
                                                        >num_rows);
   for (i = 0; i < num_pixels; i++)
   {
      if (proc_img->raw_image[i] != orig_img->raw_image[i])
```

```
            return 0;
        else
            return 1;
    }
}
```

huff.h:

```
#ifndef INCLUDE_HUFF
#define INCLUDE_HUFF

typedef struct
{
   unsigned int huffcode;
   unsigned int num_bits;
}
HUFF_DICT;

typedef struct
{
   unsigned long int frequency;     //individual frequency for leaves;
                        // sum of children's frequencies for non-leaves
   struct HUFF_NODE *lt_ptr;        //NULL for leaves
   struct HUFF_NODE *rt_ptr;        //NULL for leaves
   unsigned int pix_val;            //NULL for non-leaves
}
HUFF_NODE;

typedef struct
{
   unsigned long int frequency;     //individual frequency if ptr
                  // points to leaf; sum of children's frequencies if
                  // ptr points to non-leaves
   HUFF_NODE *huff_ptr; //points to HUFF_NODE type child or parent
}
HEAP_NODE;

//creates Huffman Dictionary--only creates codes for pixel values found
// in image
HUFF_DICT *hdcreate(unsigned long int *dist, unsigned int length);

//add new value to heap
//new_i is first empty leaf
void HeapAdd(HEAP_NODE *heap_tree, HEAP_NODE *newheap, unsigned int
                                                                  new_i);

//take top (smallest) value from heap
//last_i is last filled leaf
HEAP_NODE HeapSubtract(HEAP_NODE *heap_tree, unsigned int last_i);

//traverse tree in order (LVR traversal)
void TreeTraverse(HUFF_NODE *huffnode, HUFF_DICT *hdict, unsigned int
                                              code, unsigned int bits);

//Huffman encodes image with dictionary
//outputs compressed file "compfile_name"
void huffenco(HUFF_DICT *hdict, RAW_IMG *uncomp);

//Huffman encodes int array with dictionary
//outputs compressed file "compfile_name"
//size is length of array
```

```
void huffencoa(HUFF_DICT *hdict, int *array, unsigned long int size,
                                                        char *compfile);

//write out variable length code
//returns number of values (pixels) written out
int bitwrite(HUFF_DICT *hdict, unsigned char *raw_image, long int
                                          num_pixels, char *comp_file);

//same as bitwrite but with input of int array instead of unsigned char
//assumes values in array go from -255 to 255
int bitwritea(HUFF_DICT *hdict, int *array, long int size, char
                                                          *comp_file);

//decodes Huffman-compressed image
//outputs RAW_IMG object
struct RAW_IMG *huffdeco(char *compfile, int num_rows, int num_cols);

//decodes Huffman-compressed array
//outputs integer array
int *huffdecoa(char *compfile, long int size);

#endif
```

huff.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "rawio.h"
#include "huff.h"

//non-local variables
HUFF_NODE *huff_tree;
HUFF_NODE *top_node;

//creates Huffman Dictionary--only creates codes for pixel values found
in image
HUFF_DICT *hdcreate(unsigned long int *dist, unsigned int length)
{
   //create and initialize array of leaves for nonzero frequency pixel
   // values
   unsigned int i,j;
   unsigned int huff_i = 0;   //index for nonzero frequency values
   unsigned int num_left;           //number of frequencies left
   HEAP_NODE *heap_tree;
   HEAP_NODE first, second;
   HEAP_NODE newnode;
   HUFF_DICT *hdict;
   unsigned int size = 2*length - 1;

   //allocate memory for array
   huff_tree = malloc(size*sizeof(HUFF_NODE));
   if(huff_tree == NULL)
      printf( "Insufficient memory available\n" );

   //initialize leaves
   for(i = 0; i <= length-1; i++)
   {
      if (*(dist+i) > 0)
      {
         huff_tree[huff_i].frequency = *(dist+i);
         huff_tree[huff_i].lt_ptr = NULL;
         huff_tree[huff_i].rt_ptr = NULL;
         huff_tree[huff_i].pix_val = i;
         huff_i++;
      }
   }

   num_left = huff_i;

   //create and order array of HEAP_NODEs
   heap_tree = malloc(size*sizeof(HEAP_NODE));
   if (heap_tree == NULL)
      printf( "Insufficient memory available\n");

   for (i = 0; i < num_left; i++)
   {
      heap_tree[i].frequency = huff_tree[i].frequency;
      heap_tree[i].huff_ptr = (HUFF_NODE *)&huff_tree[i];
```

```
      HeapAdd(heap_tree,&heap_tree[i],i);
   }

   //build up Huffman tree
   while (num_left > 1)
   {
      //take two least frequent nodes (top of heap)
      first = HeapSubtract(heap_tree,num_left-1);
      num_left--;
      second = HeapSubtract(heap_tree,num_left-1);

      //create new HUFF_NODE
      huff_tree[huff_i].frequency = first.frequency+second.frequency;
      huff_tree[huff_i].lt_ptr = (struct HUFF_NODE *)first.huff_ptr;
      huff_tree[huff_i].rt_ptr = (struct HUFF_NODE *)second.huff_ptr;
      huff_tree[huff_i].pix_val = 0;

      //create new HEAP_NODE that references new HUFF_NODE
      newnode.huff_ptr = &huff_tree[huff_i];
      newnode.frequency = huff_tree[huff_i].frequency;

      //add new HEAP_NODE to heap tree
      HeapAdd(heap_tree,&newnode,num_left-1);

      //increment huff_i
      huff_i++;
   }

   //allocate memory for dictionary
   hdict = calloc(length,sizeof(HUFF_DICT));
   if(hdict == NULL)
      printf( "Insufficient memory available\n" );

   //traverse tree to find codes and place them in dictionary
   top_node = &huff_tree[huff_i-1];
   TreeTraverse(top_node,hdict,0,0);

   if (hdict->num_bits == 0)
      (hdict->num_bits)++;

   return hdict;
}

//add new value to heap
//new_i is first empty leaf
void HeapAdd(HEAP_NODE *heap_tree, HEAP_NODE *newvalue, unsigned int
                                                                  new_i)
{
   HEAP_NODE temp;
   unsigned int child_i;
   unsigned int parent_i;

   heap_tree[new_i] = *newvalue;
   child_i = new_i;
   parent_i = (child_i-1)/2;

   //initialize new HEAP_NODE
```

```
   while ((heap_tree[parent_i].frequency >
                        heap_tree[child_i].frequency)&&(child_i != 0))
   {
      temp = heap_tree[parent_i];
      heap_tree[parent_i] = heap_tree[child_i];
      heap_tree[child_i] = temp;
      child_i = parent_i;
      parent_i = (child_i-1)/2;
   }
}

//remove top (smallest) value from heap
//last_i is last filled leaf
HEAP_NODE HeapSubtract(HEAP_NODE *heap_tree, unsigned int last_i)
{
   HEAP_NODE top;
   HEAP_NODE temp;
   unsigned int parent_i = 0;
   unsigned int child_1 = (parent_i+1)*2-1;
   unsigned int child_2 = (parent_i+1)*2;
   unsigned int least_i;

   top = heap_tree[0];
   heap_tree[0] = heap_tree[last_i];

   while (((heap_tree[parent_i].frequency >
            heap_tree[child_1].frequency)||(heap_tree[parent_i].frequen
            cy > heap_tree[child_2].frequency))&&(child_1 <= last_i-1))
   {
      if ((child_1 == last_i-1)||(heap_tree[child_1].frequency <
                                          heap_tree[child_2].frequency))
         least_i = child_1;
      else
         least_i = child_2;
      temp = heap_tree[parent_i];
      heap_tree[parent_i] = heap_tree[least_i];
      heap_tree[least_i] = temp;
      parent_i = least_i;
      child_1 = (parent_i+1)*2-1;
      child_2 = (parent_i+1)*2;
   }
   return top;
}

//traverse tree in order (LVR traversal)
void TreeTraverse(HUFF_NODE *huffnode, HUFF_DICT *hdict, unsigned int
                                                  code, unsigned int bits)
{
   int pixel;
   HUFF_NODE cur;

   cur = *huffnode;
   if (huffnode->lt_ptr != NULL)
   {
      TreeTraverse((HUFF_NODE *)huffnode->lt_ptr,hdict, code << 1, bits
                                                                  + 1);
```

```
      TreeTraverse((HUFF_NODE *)huffnode->rt_ptr,hdict, (code << 1) +
                                                              1, bits + 1);
   }
   else
   {
      pixel = huffnode->pix_val;
      hdict[huffnode->pix_val].huffcode = code;
      hdict[huffnode->pix_val].num_bits = bits;
   }
}

//Huffman encodes image with dictionary
//outputs compressed file "compfile_name"
void huffenco(HUFF_DICT *hdict, RAW_IMG *uncomp)
{
   long int pixels = 0;
   pixels = bitwrite(hdict,uncomp->raw_image,(uncomp-
               >num_cols)*(uncomp->num_rows),uncomp->raw_image_name);
   if (pixels < (uncomp->num_cols)*(uncomp->num_rows))
      printf("Not all data compressed\n");
}

//Huffman encodes int array with dictionary
//outputs compressed file "compfile_name"
//size is length of array
void huffencoa(HUFF_DICT *hdict, int *array, unsigned long int size,
char *compfile)
{
   unsigned long int pixels = 0;
   pixels = bitwritea(hdict,array,size,compfile);
   if (pixels < size)
      printf("Not all data compressed\n");
}

//write out variable length code
//returns number of values (pixels) written out
int bitwrite(HUFF_DICT *hdict, unsigned char *raw_image, long int
                                          num_pixels, char *comp_file)
{
   unsigned long int huffcode;
   unsigned long int mask;
   unsigned char shift;
   unsigned int bit;
   unsigned char bit_num, bytebuffer = 0;
   int i, numwritten, buff_bits = 0;
   FILE *stream;

   //open file for writing
   stream = fopen(comp_file,"wb");
   if (stream == NULL)
      printf("The file was not opened\n");

   for (i = 0; i < num_pixels; i++)
   {
      //using dictionary code, write variable length code to buffer
      for (bit_num = 1; bit_num <= hdict[raw_image[i]].num_bits;
                                                          bit_num++)
```

```
      {
         huffcode = hdict[raw_image[i]].huffcode;
         mask = 1<<(hdict[raw_image[i]].num_bits-bit_num);
         shift = hdict[raw_image[i]].num_bits-bit_num;
         bit = (huffcode&mask)>>shift;
         bytebuffer = (bytebuffer << 1) + bit;
         buff_bits++;

         //if buffer is full, write to file
         if (buff_bits == 8)
         {
            numwritten = fwrite(&bytebuffer,1,1,stream);
            if (numwritten != 1)
               printf("Data write error\n");
            bytebuffer = 0;
            buff_bits = 0;
         }
      }
   }

   //if extra bits are left in buffer, flush by padding with zeros and
   // writing to file
   if (buff_bits != 0)
   {
      bytebuffer = bytebuffer << (8 - buff_bits);
      numwritten = fwrite(&bytebuffer,1,1,stream);
      if (numwritten != 1)
         printf("Data write error\n");
   }
   fclose(stream);

   return i;
}

//same as bitwrite but with input of int array instead of unsigned char
//assumes values in array go from -255 to 255
int bitwritea(HUFF_DICT *hdict, int *array, long int size, char
                                                              *comp_file)
{
   unsigned long int huffcode;
   unsigned long int mask;
   unsigned char shift;
   unsigned int bit;
   unsigned char bit_num, bytebuffer = 0;
   int i, numwritten, buff_bits = 0;
   FILE *stream;

   //open file for writing
   stream = fopen(comp_file,"wb");
   if (stream == NULL)
      printf("The file was not opened\n");

   for (i = 0; i < size; i++)
   {
      //using dictionary code, write variable length code to buffer
      for (bit_num = 1; bit_num <= hdict[255+array[i]].num_bits;
                                                            bit_num++)
```

```
      {
         huffcode = hdict[255+array[i]].huffcode;
         mask = 1<<(hdict[255+array[i]].num_bits-bit_num);
         shift = hdict[255+array[i]].num_bits-bit_num;
         bit = (huffcode&mask)>>shift;
         bytebuffer = (bytebuffer << 1) + bit;
         buff_bits++;

         //if buffer is full, write to file
         if (buff_bits == 8)
         {
            numwritten = fwrite(&bytebuffer,1,1,stream);
            if (numwritten != 1)
               printf("Data write error\n");
            bytebuffer = 0;
            buff_bits = 0;
         }
      }
   }

   //if extra bits are left in buffer, flush by padding with zeros and
   // writing to file
   if (buff_bits != 0)
   {
      bytebuffer = bytebuffer << (8 - buff_bits);
      numwritten = fwrite(&bytebuffer,1,1,stream);
      if (numwritten != 1)
         printf("Data write error\n");
   }
   fclose(stream);

   return i;
}

//decodes Huffman-compressed image
//outputs RAW_IMG object
struct RAW_IMG *huffdeco(char *compfile, int num_rows, int num_cols)
{
   unsigned char direction, mask;
   long int numread;
   unsigned int buff_bits = 0;
   unsigned char compbuffer = 0;
   long int num_pixels = 0;
   HUFF_NODE *huffnode;
   FILE *stream;
   RAW_IMG *img = malloc(sizeof(RAW_IMG));

   if (img == NULL)
      printf( "Insufficient memory available\n" );

   //open compressed file for reading
   stream = fopen(compfile,"rb");
   if(stream == NULL)
      printf( "The file was not opened\n" );

   //allocate memory for image
   img->raw_image = calloc(num_rows*num_cols,1);
```

```
   if(img->raw_image == NULL)
      printf( "Insufficient memory available\n" );

   while (num_pixels < num_rows*num_cols)
   {
      huffnode = top_node;
      while (huffnode->lt_ptr != NULL)
      {
         if (buff_bits == 0)
         {
            numread = fread(&compbuffer,1,1,stream);
            if (numread != 1)
            {
               printf("Data read error numread = %d\n",numread);
               printf("  feof: %d\n  ferror: %d\n", feof(stream),
                                                    ferror(stream));
            }
            buff_bits++;
         }
         mask = 1 << (8-buff_bits);
         direction = (compbuffer&mask) >> (8-buff_bits);
         if (direction == 0)
            huffnode = (HUFF_NODE *)huffnode->lt_ptr;
         else
            huffnode = (HUFF_NODE *)huffnode->rt_ptr;
         buff_bits = (buff_bits + 1) % 9;
      }
      img->raw_image[num_pixels] = (unsigned char)huffnode->pix_val;
      num_pixels++;
   }
   fclose(stream);

   //set image characteristics
   img->num_cols = num_cols;
   img->num_rows = num_rows;
   img->raw_image_name = compfile;
   img->code = 0;              //assume binary

   return (struct RAW_IMG *)img;
}

//decodes Huffman-compressed array
//outputs integer array
int *huffdecoa(char *compfile, long int size)
{
   unsigned char direction, mask;
   long int numread;
   unsigned int buff_bits = 0;
   unsigned char compbuffer = 0;
   long int num_pixels = 0;
   HUFF_NODE *huffnode;
   FILE *stream;
   int *array = calloc(size,sizeof(int));

   if (array == NULL)
      printf( "Insufficient memory available\n" );
```

```
    //open compressed file for reading
    stream = fopen(compfile,"rb");
    if(stream == NULL)
       printf( "The file was not opened\n" );

    while (num_pixels < size)
    {
       huffnode = top_node;
       while (huffnode->lt_ptr != NULL)
       {
          if (buff_bits == 0)
          {
             numread = fread(&compbuffer,1,1,stream);
             if (numread != 1)
                printf("Data read error numread = %d\n",numread);
             buff_bits++;
          }
          mask = 1 << (8-buff_bits);
          direction = (compbuffer&mask) >> (8-buff_bits);
          if (direction == 0)
             huffnode = (HUFF_NODE *)huffnode->lt_ptr;
          else
             huffnode = (HUFF_NODE *)huffnode->rt_ptr;
          buff_bits = (buff_bits + 1) % 9;
       }
       array[num_pixels] = (int)(huffnode->pix_val - 255);
       num_pixels++;
    }
    fclose(stream);

    return array;
}
```

LZW.h:

```
#ifndef INCLUDE_LZW
#define INCLUDE_LZW

typedef struct
{
   unsigned int address;      //begins at 0; variable number of bits
                              // used to encode
   unsigned int pix_val;      //0 to 255 (or more for int arrays)
   struct LZW_NODE *less_ptr; //points to node with smaller pixel value
                              // (or NULL)
   struct LZW_NODE *more_ptr; //points to node with greater pixel value
                              // (or NULL)
   struct LZW_NODE *seq_ptr;  //points to next node in sequence (used
                  // if current pixel value is in sequence) or NULL
}
LZW_NODE;

typedef struct
{
   unsigned int *entry;
   unsigned int entry_num;
}
DECODE_DICT;

//LZW encodes using 8 bit RAW_IMG object and histogram
//codes are only created if pixel value is in histogram; histogram must
// be appended to compressed file
//returns number of entries in dictionary
unsigned long int LZWenco(char *compfile, RAW_IMG *raw_image, unsigned
                                                  long int *histogram);

//LZW encodes using integer array and histogram
//codes are only created if pixel value is in histogram; histogram must
// be appended to compressed file
//returns number of entries in dictionary
//size is number of entries in array
unsigned long int LZWencoa(char *compfile, int *array, unsigned long
                                int size, unsigned long int *histogram);

//searches tree for pixel value and adds new node if it doesn't exist
//returns node with pixel value if it exists, or old node if it had to
// be added
LZW_NODE *searchtree(LZW_NODE *node, unsigned int pixel);

//adds new node to binary tree at seq_ptr
void addnode(LZW_NODE *node, unsigned int pixel, unsigned long int
                                                              address);

//reads and decodes compressed LZW file given histogram, number of
// rows, columns, and dictionary entries
RAW_IMG *LZWdeco(char *compfile, unsigned long int *hist, int num_rows,
int num_cols, unsigned long int dict_entries);

//same as LZWdeco, but generates int array instead of RAW_IMG
```

```
int *LZWdecoa(char *compfile, unsigned long int *hist, unsigned long
                              int size, unsigned long int dict_entries);

#endif
```

LZW.c:

```
#include "rawio.h"
#include "LZW.h"
#include <stdlib.h>
#include <malloc.h>
#include <stdio.h>
#include "bitplane.h"
#include "stats.h"

//LZW encodes using 8 bit RAW_IMG object and histogram
//codes are only created if pixel value is in histogram; histogram must
// be appended to compressed file
//returns number of entries in dictionary
unsigned long int LZWenco(char *compfile, RAW_IMG *raw_image, unsigned
                                                        long int *hist)
{
   FILE *bytestream;
   unsigned char num_bits, found;
   unsigned int i, pixel, num_dict;
   unsigned long int num_pixels = (raw_image->num_cols)*(raw_image-
                                                >num_rows), codewritten;
   unsigned long int cur_address = 0;
   BYTE_BUFFER *tempbuffer = malloc(sizeof(BYTE_BUFFER));
   LZW_NODE *node, *child, *dict = malloc(256*sizeof(LZW_NODE));

   //assign initial nodes to pixels found in histogram
   for (i = 0; i < 256; i++)
   {
      if (hist[i])
      {
         dict[i].pix_val = i;
         dict[i].seq_ptr = NULL;
         dict[i].less_ptr = NULL;
         dict[i].more_ptr = NULL;
         dict[i].address = cur_address;
         cur_address++;
      }
   }
   cur_address--;

   i = 0;

   //open file for writing, which is done concurrently with dictionary
   // generation
   bytestream = openwritebuffer(compfile, tempbuffer);

   while (i < num_pixels)
   {
      pixel = raw_image->raw_image[i];    //current pixel value
      node = &(dict[pixel]);  //set node to parent node in dictionary
      child = node->seq_ptr;  //set child to node pointed to by parent
      if (!child)
         //if no child exists (yet), look at the next pixel
         pixel = raw_image->raw_image[++i];
      found = 0;
```

```
      while (child)
      //find node with last matching pixel
      {
         found = 0;
         pixel = raw_image->raw_image[++i];     //increment pixel
         child = searchtree(child,pixel);       //search for pixel
         if (child)
            //if pixel found
            {
               node = child;
               child = node->seq_ptr;
               found = 1;
            }
         }
         if (found)
            pixel = raw_image->raw_image[++i];  //go to next pixel
         num_bits = 0;
         num_dict = cur_address;
         while(num_dict)      //check number of bits in current address
         {
            num_bits++;
            num_dict = num_dict >> 1;
         }

         //write node address with last matching pixel
         codewritten = write_bits(bytestream, tempbuffer, &(node-
                                                >address), 1, num_bits);
         if (codewritten != 1)
            printf("Write code array error\n");

         //add new node
         if (i < num_pixels)
            addnode(&node, pixel, ++cur_address);
   }

   //flush any left over bits in buffer
   num_bits = flushwritebuffer(bytestream, tempbuffer);

   return cur_address + 1;
}

//LZW encodes using integer array and histogram
//codes are only created if pixel value is in histogram; histogram must
// be appended to compressed file
//returns number of entries in dictionary
//size is number of entries in array
unsigned long int LZWencoa(char *compfile, int *array, unsigned long
                                int size, unsigned long int *histogram)
{
   FILE *bytestream;
   unsigned char num_bits, found;
   unsigned int i, pixel, num_dict;
   unsigned long int codewritten;
   unsigned long int cur_address = 0;
   BYTE_BUFFER *tempbuffer = malloc(sizeof(BYTE_BUFFER));
   LZW_NODE *node, *child, *dict = malloc(512*sizeof(LZW_NODE));
```

```
//assign initial nodes to values found in histogram
for (i = 0; i < 511; i++)
{
     if (histogram[i])
     {
         dict[i].pix_val = i;
         dict[i].seq_ptr = NULL;
         dict[i].less_ptr = NULL;
         dict[i].more_ptr = NULL;
         dict[i].address = cur_address;
         cur_address++;
     }
}
cur_address--;

i = 0;

//open file for writing, which is done concurrently with dictionary
// generation
bytestream = openwritebuffer(compfile, tempbuffer);
while (i < size)
{
   pixel = 255+array[i];               //current pixel value
   node = &(dict[pixel]);  //set node to parent node in dictionary
   child = node->seq_ptr;  //set child to node pointed to by parent
   if (!child)
      pixel = 255+array[++i];
   found = 0;
   while (child)
   //find node with last matching pixel
   {
      found = 0;
      pixel = 255+array[++i];                //increment pixel
      child = searchtree(child,pixel);       //search for pixel
      if (child)
      //if pixel found
      {
         node = child;
         child = node->seq_ptr;
         found = 1;
      }
   }
   if (found)
      pixel = 255+array[++i];          //go to next pixel
   num_bits = 0;
   num_dict = cur_address;
   while(num_dict)
   //check number of bits in current address
   {
      num_bits++;
      num_dict = num_dict >> 1;
   }

   //write node address with last matching pixel
   codewritten = write_bits(bytestream, tempbuffer, &(node-
                                          >address), 1, num_bits);
   if (codewritten != 1)
```

```
            printf("Write code array error\n");

         //add new node
         if (i < size)
            addnode(&node, pixel, ++cur_address);
      }

      //flush any left over bits in buffer
      num_bits = flushwritebuffer(bytestream, tempbuffer);

      return cur_address + 1;
   }

   //searches tree for pixel value and adds new node if it doesn't exist
   //returns node with pixel value if it exists, or old node if it had to
   // be added
   LZW_NODE *searchtree(LZW_NODE *node, unsigned int pixel)
   {
      while (node)
      {
         if (pixel == node->pix_val)
            return node;
         else
         {
            if (pixel < node->pix_val)
               node = node->less_ptr;
            else
               node = node->more_ptr;
         }
      }

      return NULL;
   }

   //adds new node to binary tree at seq_ptr
   void addnode(LZW_NODE **node, unsigned int pixel, unsigned long int
                                                                  address)
   {
      unsigned char newnode = 0;

      //if node does not have seq_ptr
      if (!(*node)->seq_ptr)
      {
         //generate new node pointed to by seq_ptr
         (*node)->seq_ptr = malloc(sizeof(LZW_NODE));

         //follow seq_ptr to new mode
         *node = (*node)->seq_ptr;
         if (!*node)
            printf("Insufficient memory available");
      }
      //if node does have seq_ptr
      else
      {
         //follow seq_ptr to next node
         (*node) = (*node)->seq_ptr;
```

```
   //while the new node does not exist
   while (!newnode)
   {
      //if pixel value of new node is less than pixel value of
      // current node
      if (pixel < (*node)->pix_val)
      {
         //if pointer to lesser pixel value exists
         if ((*node)->less_ptr)

            //follow less_ptr to next node
            *node = (*node)->less_ptr;

         //if pointer to lesser pixel value does not exist
         else
         {
            //generate new node pointed to by less_ptr
            (*node)->less_ptr = malloc(sizeof(LZW_NODE));

            //follow less_ptr to new node
            *node = (*node)->less_ptr;
            if (!*node)
               printf("Insufficient memory available");

            //indicate new node has been generated
            newnode = 1;
         }
      }
      //if pixel value of new node is more than pixel value of
      // current node
      else
      {
         //if pointer to greater pixel value exists
         if ((*node)->more_ptr)

                        //follow more_ptr to next node
                        *node = (*node)->more_ptr;

         //if pointer to greater pixel value does not exist
         else
         {
            //generate new node pointed to by more_ptr
            (*node)->more_ptr = malloc(sizeof(LZW_NODE));
            //follow more_ptr to new node
            *node = (*node)->more_ptr;
            if (!*node)
               printf("Insufficient memory available");

            //indicate new node has been generated
            newnode = 1;
         }
      }
   }
}

//set node characteristics
(*node)->address = address;
```

```
   (*node)->pix_val = pixel;
   (*node)->less_ptr = NULL;
   (*node)->more_ptr = NULL;
   (*node)->seq_ptr = NULL;

   return;
}

//reads and decodes compressed LZW file given histogram, number of
// rows, columns, and dictionary entries
RAW_IMG *LZWdeco(char *compfile, unsigned long int *hist, int num_rows,
                          int num_cols, unsigned long int dict_entries)
{
   unsigned long int num_bits, j, i, num_dict, numread, cur_address =
                        0, *code = calloc(2, sizeof(unsigned long int));
   FILE *readstream;
   BYTE_BUFFER *readbuffer = malloc(sizeof(BYTE_BUFFER));
   RAW_IMG *img = malloc(sizeof(RAW_IMG));
   DECODE_DICT *dictionary = malloc(dict_entries*sizeof(DECODE_DICT));

   //allocate memory for image
   img->raw_image = calloc(num_rows*num_cols, 1);
   if ((!img)||(!img->raw_image)||(!code)||(!dictionary))
      printf("Insufficient memory available\n");

   //assign initial nodes to values found in histogram
   for (i = 0; i < 256; i++)
   {
      if (hist[i])
      {
         dictionary[cur_address].entry_num = 1;
         dictionary[cur_address].entry =
                                malloc(dictionary[cur_address].entry_num
                                *sizeof(unsigned int));
         if (!dictionary[cur_address].entry)
            printf("Insufficient memory\n");
         dictionary[cur_address].entry[0] = i;
         cur_address++;
      }
   }

   cur_address;

   i = 0;

   //open compressed file for reading
   readstream = openreadbuffer(compfile, readbuffer);

   //calculate number of bits in first dictionary entry
   num_bits = 0;
   num_dict = cur_address-1;
   while(num_dict)
   {
      num_bits++;
      num_dict = num_dict >> 1;
   }
```

```
//read in first entry
numread = read_bits(readstream, readbuffer, code, 1, num_bits);
if (numread != 1)
   printf("Error reading data\n");

code[1] = code[0];

while (i < num_rows*num_cols)
{
   //calculate number of bits in entry
   num_bits = 0;
   num_dict = cur_address;
   while(num_dict)
   {
      num_bits++;
      num_dict = num_dict >> 1;
   }

   code[0] = code[1];      //shift address

   if (cur_address < dict_entries)
      numread = read_bits(readstream, readbuffer, &code[1], 1,
                                                          num_bits);
   if (numread != 1)
      printf("Error reading data\n");

   dictionary[cur_address].entry_num =
                                    dictionary[code[0]].entry_num+1;

   //allocate memory for dictionary entry
   dictionary[cur_address].entry =
                         malloc((dictionary[cur_address].entry_num)*
                         sizeof(unsigned int));
   if (!(dictionary[cur_address].entry))
      printf("Insufficient memory\n");

   for (j = 0; j < dictionary[code[0]].entry_num; j++)
      dictionary[cur_address].entry[j] =
                                       dictionary[code[0]].entry[j];

   dictionary[cur_address].entry[j] = dictionary[code[1]].entry[0];

   //write pixel value to raw image array
   for (j = 0; j < dictionary[code[0]].entry_num; j++)
   {
      img->raw_image[i] = (unsigned
                                 char)(dictionary[code[0]].entry[j]);
      i++;
   }
   cur_address++;
}
fclose(readstream);

//set image characteristics
img->num_cols = num_cols;
img->num_rows = num_rows;
img->raw_image_name = "LZW.dec";
```

```
   img->code = 0;                    //assume binary coding

   return img;
}

//same as LZWdeco, but generates int array instead of RAW_IMG
int *LZWdecoa(char *compfile, unsigned long int *hist, unsigned long
                              int size, unsigned long int dict_entries)
{
   unsigned long int num_bits, j, i, num_dict, numread, cur_address =
                        0, *code = calloc(2, sizeof(unsigned long int));
   FILE *readstream;
   BYTE_BUFFER *readbuffer = malloc(sizeof(BYTE_BUFFER));
   int *array = malloc(size*sizeof(int));
   DECODE_DICT *dictionary = malloc(dict_entries*sizeof(DECODE_DICT));

   if ((!array)||(!code)||(!dictionary))
      printf("Insufficient memory available\n");

   //assign initial nodes to values found in histogram
   for (i = 0; i < 511; i++)
   {
      if (hist[i])
      {
         dictionary[cur_address].entry_num = 1;
         dictionary[cur_address].entry =
                              malloc(dictionary[cur_address].entry_num*
                              sizeof(unsigned int));
         if (!dictionary[cur_address].entry)
            printf("Insufficient memory\n");
         dictionary[cur_address].entry[0] = i;
         cur_address++;
      }
   }
   cur_address;

   i = 0;

   //open compressed file for reading
   readstream = openreadbuffer(compfile, readbuffer);

   //calculate number of bits in first dictionary entry
   num_bits = 0;
   num_dict = cur_address-1;
   while(num_dict)
   {
      num_bits++;
      num_dict = num_dict >> 1;
   }

   //read in first entry
   numread = read_bits(readstream, readbuffer, code, 1, num_bits);
   if (numread != 1)
      printf("Error reading data\n");

   code[1] = code[0];
```

```
   while (i < size)
   {
      //calculate number of bits in entry
      num_bits = 0;
      num_dict = cur_address;
      while(num_dict)
      {
         num_bits++;
         num_dict = num_dict >> 1;
      }

      code[0] = code[1];      //shift address

      if (cur_address < dict_entries)
         numread = read_bits(readstream, readbuffer, &code[1], 1,
                                                              num_bits);
      if (numread != 1)
         printf("Error reading data\n");

      dictionary[cur_address].entry_num =
                                       dictionary[code[0]].entry_num+1;

      //allocate memory for dictionary entry
      dictionary[cur_address].entry =
                           malloc((dictionary[cur_address].entry_num)*
                           sizeof(unsigned int));
      if (!(dictionary[cur_address].entry))
         printf("Insufficient memory\n");

      for (j = 0; j < dictionary[code[0]].entry_num; j++)
         dictionary[cur_address].entry[j] =
                                          dictionary[code[0]].entry[j];

      dictionary[cur_address].entry[j] = dictionary[code[1]].entry[0];

      //write pixel value to integer array
      for (j = 0; j < dictionary[code[0]].entry_num; j++)
      {
         array[i] = (int)dictionary[code[0]].entry[j]-255;
         i++;
      }

      cur_address++;
   }
   fclose(readstream);

   return array;
}
```

predictive.h:

```
#ifndef INCLUDE_PREDICTIVE
#define INCLUDE_PREDICTIVE

#include <math.h>

//encodes a RAW_IMG using weighted predictive coding
//weights all adjacent (previous) pixels--diagonal pixels by weighting
// factor
//for edges and corners, uses only those pixels available
//returns pointer to array of signed integers corresponding to errors
int *predenco(RAW_IMG *img, float weight);

//rounds number to nearest integer
int round(double value);

//decoder for above predictive encoder
//RAW_IMG is input to maintain file data
RAW_IMG *preddeco(int *error, float weight, RAW_IMG *img);

#endif
```

predictive.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
#include "rawio.h"
#include "predictive.h"
#include "bitplane.h"
#include "binarycomp.h"

//encodes a RAW_IMG using weighted predictive coding
//weights all adjacent (previous) pixels--diagonal pixels by weighting
// factor
//for edges and corners, uses only those pixels available
//returns pointer to array of signed integers corresponding to errors
int *predenco(RAW_IMG *img, float weight)
{
   long int i;
   int j,k, up, upright, upleft, left;
   int prediction;

   //create error array
   int *error = malloc(sizeof(int)*(img->num_cols)*(img->num_rows));
   if (!error)
      printf("Insufficient memory available\n");

   for (i = 0; i < (img->num_cols)*(img->num_rows); i++)
   {
      j = i/(img->num_cols);      //row number
      k = i % (img->num_cols);    //column number

      if (j == 0)                          //top row
      {
         if (k == 0)                       //top left corner
         {                                 //can't predict anything
            prediction = 0;
         }
         else                              //rest of top row
         {                           //predict based on pixel to left
            left = i-1;
            prediction = img->raw_image[left];
         }
      }
      else if (k == 0)              //left row non-top corner
      {
      //predict based on pixel above, and pixel above right
         up = (j-1)*(img->num_cols)+k;
         upright = up+1;
         prediction = round((img->raw_image[up] + (img-
                                  >raw_image[upright])*weight)/2.0);
      }
      else if (k == img->num_cols - 1)    //right row non-top corner
      {            //predict based on upper left, left, and upper pixel
         up = (j-1)*(img->num_cols)+k;
         upleft = up-1;
```

```
         left = i-1;
         prediction = round((img->raw_image[up]+(img-
                >raw_image[upleft])*weight+img->raw_image[left])/3.0);

      }
      else                                   //center of image
      {
      //predict based on four pixels to left and above center pixel
         up = (j-1)*(img->num_cols)+k;
         upleft = up-1;
         upright = up+1;
         left = i-1;
         prediction = round((img->raw_image[up]+img-
                     >raw_image[left]+weight*(img-
                     >raw_image[upleft]+img->raw_image[upright]))/4.0);
      }
      error[i] = img->raw_image[i] - prediction;
   }

   return error;
}

//rounds number to nearest integer
int round(double value)
{
   int rvalue = (int)value;

   if (value >= 0)     //positive input
   {
      if (value >= rvalue + 0.5)
         rvalue++;
   }
   else                //negative input
   {
      if (value <= rvalue - 0.5)
         rvalue--;
   }

    return rvalue;
}

//decoder for above predictive encoder
//RAW_IMG is input to maintain file data
RAW_IMG *preddeco(int *error, float weight, RAW_IMG *img)
{
   long int i;
   unsigned int j,k, up, upright, upleft, left;
   int prediction;

   //allocate memory for decompressed image structure
   RAW_IMG *rtn_img = malloc(sizeof(RAW_IMG));
   if (rtn_img == NULL)
      printf( "Insufficient memory available\n" );

   //allocate memory for decompressed image
   rtn_img->raw_image = calloc(img->num_rows*img->num_cols,1);
   if(rtn_img->raw_image == NULL)
```

```
{
   printf( "Insufficient memory available\n" );
}

//copy image parameters
rtn_img->code = img->code;
rtn_img->num_cols = img->num_cols;
rtn_img->num_rows = img->num_rows;
rtn_img->raw_image_name = img->raw_image_name;

//predict pixel values based on preceding values
for (i = 0; i < (rtn_img->num_cols)*(rtn_img->num_rows); i++)
{
   j = i / (rtn_img->num_cols);  //row number
   k = i % (rtn_img->num_cols);  //column number

   if (j == 0)                                   //top row
   {
      if (k == 0)                                //top left corner
      {                                    //can't predict anything
         prediction = 0;
      }
      else                                       //rest of top row
      {                              //predict based on pixel to left
         left = i-1;
         prediction = rtn_img->raw_image[left];
      }
   }
   else if (k == 0)                       //left row non-top corner
   {           //predict based on pixel above, and pixel above right
      up = (j-1)*(rtn_img->num_cols)+k;
      upright = up+1;
      prediction = round((rtn_img->raw_image[up] + (rtn_img-
                                 >raw_image[upright])*weight)/2.0);
   }
   else if (k == rtn_img->num_cols - 1)   //right row non-top corner
   {           //predict based on upper left, left, and upper pixel
      up = (j-1)*(rtn_img->num_cols)+k;
      upleft = up-1;
      left = i-1;
      prediction = round((rtn_img->raw_image[up]+(rtn_img-
          >raw_image[upleft])*weight+rtn_img->raw_image[left])/3.0);

   }
   else                                          //center of image
   {   //predict based on four pixels to left and above center pixel
      up = (j-1)*(rtn_img->num_cols)+k;
      upleft = up-1;
      upright = up+1;
      left = i-1;
      prediction = round((rtn_img->raw_image[up]+rtn_img-
             >raw_image[left]+weight*(rtn_img-
             >raw_image[upleft]+rtn_img->raw_image[upright]))/4.0);
   }

   //add error to prediction to get original pixel value
   rtn_img->raw_image[i] = (unsigned char)(error[i] + prediction);
```

```
    }

    return (struct RAW_IMG *)rtn_img;
}
```

bitplane.h:

```
#ifndef INCLUDE_BITPLANE
#define INCLUDE_BITPLANE

typedef struct
{
   unsigned char *bp;               //pointer to individual bitplane
   unsigned int num_bytes;          //number of bytes in plane--extra
                                    // bits are 0
   unsigned char comptype;          //indicates type of compression -
                                    // 0 for none
   unsigned char code;              //0 for binary, 1 for gray
}
PLANE;

typedef struct
{
   RAW_IMG *raw_image;
   PLANE *bplane[8];                //array of 8 pointers to PLANE
                                    // objects (0: MS to 7: LS)
}
BIT_PLANE;

//divides an 8-bit raw image into a BIT_PLANE
BIT_PLANE *sliceimage(RAW_IMG *raw_image);

//reconstructs RAW_IMG from uncompressed BIT_PLANE
//overwrites any raw image previously existing in RAW_IMG
//assumes RAW_IMG contains a filename and numbers of columns and rows
//if bit plane is not uncompressed, returns NULL and prints error
// message
RAW_IMG *joinimage(BIT_PLANE *bpobject);

//writes to file bit planes from MS to LS
//first four bytes in each file are file header
//first two bytes in file header are number of rows
//next two bytes in file header are number of columns
//first four bytes in each plane are header
//first plane byte indicates type of compression (0 if none)
//next three plane bytes indicate are number of bytes in plane
//last byte in each plane is padded with zeros if necessary
//returns total number of bytes written out
unsigned long int writeBP(BIT_PLANE *bpobject);

//reads in bit plane from file, creating and returning BIT_PLANE object
//num_cols and num_rows are set with header information, but
// pointers to other members of RAW_IMG structure are set to null
BIT_PLANE *readBP(char *filename);

//turns BIT_PLANE into RAW_IMG format
RAW_IMG *bptoimg(PLANE *bplane);

//turns RAW_IMG format into BIT_PLANE
PLANE *imgtobp(RAW_IMG *img);
```

```
#endif
```

bitplane.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "rawio.h"
#include "bitplane.h"
#include "binarycomp.h"

//divides an 8-bit raw image into a BIT_PLANE
BIT_PLANE *sliceimage(RAW_IMG *raw_image)
{
   BIT_PLANE *plane = malloc(sizeof(BIT_PLANE));
   unsigned long int mask;
   unsigned char shift, j, bit, bitcount = 0, bpbuffer[8], k;
   unsigned long int pixel = 0, num_pixels = (raw_image-
                                    >num_cols*raw_image->num_rows), i;
   unsigned int num_bytes = num_pixels/8;

   plane->raw_image = raw_image;

   if ((num_pixels % 8) != 0)
      num_bytes++;

   //set up BIT_PLANE structure
   for (i = 0; i <= 7; i++)
   {
      plane->bplane[i] = malloc(sizeof(PLANE));
      if (!(plane->bplane[i]))
         printf("Insufficient memory available\n");

      plane->bplane[i]->num_bytes = num_bytes;
      plane->bplane[i]->comptype = 0;     //no compression by default
      plane->bplane[i]->bp = calloc(num_bytes,1);
   }

   for (j = 0; j < 8; j++)
      bpbuffer[j] = 0;

   for (i = 0; i < num_bytes; i++)
   {
      bitcount = 0;

      while (bitcount < 8)
      {
         for (k = 0; k < 8; k++)
         {
            shift = 7 - k;
            mask = 1 << shift;
            if (pixel < num_pixels)
               bit = ((raw_image->raw_image[pixel])&(unsigned
                                                   char)mask)>>shift;
            else
               bit = 0;
            bpbuffer[k] = (bpbuffer[k] << 1) + bit;
         }
```

```
            bitcount++;
            pixel++;
         }

         for (j = 0; j < 8; j++)
         {
            plane->bplane[j]->bp[i] = bpbuffer[j];
            bpbuffer[j] = 0;
         }
      }

      return plane;
}

//reconstructs RAW_IMG from uncompressed BIT_PLANE
//overwrites any raw image previously existing in RAW_IMG
//assumes RAW_IMG contains a filename and numbers of columns and rows
//if bit plane is not uncompressed, returns NULL and prints error
// message
RAW_IMG *joinimage(BIT_PLANE *bpobject)
{
   unsigned long int i, pixel = 0, numpixels = (bpobject->raw_image-
                              >num_cols)*(bpobject->raw_image->num_rows);
   unsigned char bpbuffer, mask, shift, bit, j, k;

   //check for compression
   for (j = 0; j < 8; j++)
   {
      if (bpobject->bplane[j]->comptype != 0)
      {
         printf("Cannot join compressed bit plane object\n");
         return NULL;
      }
   }

   //allocate memory for raw image
   bpobject->raw_image->raw_image = calloc(numpixels,1);
   if (!(bpobject->raw_image->raw_image))
      printf("Insufficient memory available\n");

   //read in raw image
   for (i = 0; i < bpobject->bplane[0]->num_bytes; i++)
   {
      bpbuffer = 0;

      for (j = 0; j < 8; j++)
      {
         for (k = 0; k < 8; k++)
         {
            shift = 7 - j;
            mask = 1 << shift;
            bit = ((bpobject->bplane[k]->bp[i])&mask)>>shift;
            bpbuffer = (bpbuffer << 1) + bit;
         }

         if (pixel < numpixels)
            bpobject->raw_image->raw_image[pixel] = bpbuffer;
```

```
            pixel++;
         }
      }

      return bpobject->raw_image;
   }

   //writes to file bit planes from MS to LS
   //first four bytes in each file are file header
   //first two bytes in file header are number of rows
   //next two bytes in file header are number of columns
   //first four bytes in each plane are header
   //first plane byte indicates type of compression (0 if none)
   //next three plane bytes indicate are number of bytes in plane
   //last byte in each plane is padded with zeros if necessary
   //returns total number of bytes written out
   unsigned long int writeBP(BIT_PLANE *bpobject)
   {
      FILE *bpstream;
      unsigned long int numwritten, totalwritten = 0;
      unsigned char header[4], i;

      bpstream = fopen("bpfile","wb");

      //create file header
      header[0] = ((bpobject->raw_image->num_rows)&(255 << 8)) >> 8;
      header[1] = ((bpobject->raw_image->num_rows)&(255));
      header[2] = ((bpobject->raw_image->num_cols)&(255 << 8)) >> 8;
      header[3] = ((bpobject->raw_image->num_cols)&(255));

      //write out header
      numwritten = fwrite(header,1,4,bpstream);
      if (numwritten < 4)
         printf("Not all data written\n");
      totalwritten += numwritten;

      //for each plane
      for (i = 0; i < 8; i++)
      {
         //create header
         header[0] = bpobject->bplane[i]->comptype;
         header[1] = ((bpobject->bplane[i]->num_bytes)&(255 << 16)) >> 16;
         header[2] = ((bpobject->bplane[i]->num_bytes)&(255 << 8)) >> 8;
         header[3] = ((bpobject->bplane[i]->num_bytes)&(255));

         //write out header
         numwritten = fwrite(header,1,4,bpstream);
         if (numwritten < 4)
            printf("Not all data written\n");
         totalwritten += numwritten;

         //write out information in bitplane [i]
         numwritten = fwrite(bpobject->bplane[i]->bp, 1, bpobject-
                                        >bplane[i]->num_bytes, bpstream);
         if (numwritten < bpobject->bplane[i]->num_bytes)
            printf("Not all data written\n");
         totalwritten += numwritten;
```

```
    }
    fclose(bpstream);
    return totalwritten;
}

//reads in bit plane from file, creating and returning BIT_PLANE object
//num_cols and num_rows are set with header information, but
// pointers to other members of RAW_IMG structure are set to null
BIT_PLANE *readBP(char *filename)
{
    BIT_PLANE *plane = malloc(sizeof(BIT_PLANE));
    FILE *bpstream;
    unsigned long int numread;
    unsigned char header[4], i;

    plane->raw_image = malloc(sizeof(RAW_IMG));

    if ((!plane)||(!plane->raw_image))
        printf("Insufficient memory available\n");

    //put information into raw_image
    plane->raw_image->raw_image_name = filename;
    plane->raw_image->raw_image = NULL;

    //open file
    bpstream = fopen(filename, "rb");
    if(bpstream == NULL)
        printf( "The file was not opened\n" );

    //read file header
    numread = fread(header,1,4,bpstream);
    if (numread < 4)
        printf("Not all data read\n");

    //put row and column information into raw_image
    plane->raw_image->num_rows = header[0];
    plane->raw_image->num_rows = (plane->raw_image->num_rows << 8) +
                                                        header[1];
    plane->raw_image->num_cols = header[2];
    plane->raw_image->num_cols = (plane->raw_image->num_cols << 8) +
                                                        header[3];

    //for each plane
    for (i = 0; i < 8; i++)
    {
        //create memory for PLANE object
        plane->bplane[i] = malloc(sizeof(PLANE));
        if (!(plane->bplane[i]))
            printf("Insufficient memory available\n");

        //read plane header
        numread = fread(header,1,4,bpstream);
        if (numread < 4)
            printf("Not all data read\n");

        //get information from header
        plane->bplane[i]->comptype = header[0];
```

```
      plane->bplane[i]->num_bytes = header[1];
      plane->bplane[i]->num_bytes = (plane->bplane[i]->num_bytes << 8)
                                                          + header[2];
      plane->bplane[i]->num_bytes = (plane->bplane[i]->num_bytes << 8)
                                                          + header[3];

      //create memory for bit plane
      plane->bplane[i]->bp = calloc(plane->bplane[i]->num_bytes,1);
      if (!(plane->bplane[i]->bp))
         printf("Insufficient memory available\n");

      //read bit planes
      numread = fread(plane->bplane[i]->bp,1,plane->bplane[i]-
                                                  >num_bytes,bpstream);
      if (numread < plane->bplane[i]->num_bytes)
          printf("Not all data read\n");
   }
   return plane;
}

//turns BIT_PLANE into RAW_IMG format
RAW_IMG *bptoimg(PLANE *bplane)
{
   RAW_IMG *img = malloc(sizeof(RAW_IMG));
   img->code = bplane->code;
   img->num_cols = bplane->num_bytes;
   img->num_rows = 1;
   img->raw_image_name = "BitPlaneImg";
   img->raw_image = bplane->bp;
   return img;
}

//turns RAW_IMG format into BIT_PLANE
PLANE *imgtobp(RAW_IMG *img)
{
   PLANE *bplane = malloc(sizeof(PLANE));
   bplane->bp = img->raw_image;
   bplane->code = img->code;
   bplane->num_bytes = img->num_cols*img->num_rows;
   return bplane;
}
```

binarycomp.h:

```
#ifndef INCLUDE_BINARYCOMP
#define INCLUDE_BINARYCOMP

#include "bitplane.h"


//nonzero encoder records position of nonzero bytes
//code is set of four bytes:
//first three are position of nonzero byte (starting at index 0)
//last is byte value
PLANE *nzcode(PLANE *rawplane);

//nonzero decoder
//num_bytes is uncompressed number of bytes in plane
PLANE *nzdecode(PLANE *compplane, unsigned int num_bytes);

#endif
```

binarycomp.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include "rawio.h"
#include "bitplane.h"
#include "binarycomp.h"

//nonzero encoder records position of nonzero bytes
//code is set of four bytes:
//first three are position of nonzero byte (starting at index 0)
//last is byte value
PLANE *nzcode(PLANE *rawplane)
{
   unsigned int compbytes = 0,i;

   //allocate memory for compressed PLANE
   PLANE *compplane = malloc(sizeof(PLANE));
   if (!compplane)
      printf("Insufficient memory\n");

   //modify compression byte
   compplane->comptype = 1;

   //allocate memory for compressed PLANE bit plane
   compplane->bp = malloc(4*(rawplane->num_bytes));
   if (!(compplane->bp))
      printf("Insufficient memory\n");

   //record position of nonzero bytes
   for (i = 0; i < rawplane->num_bytes; i++)
   {
      if (rawplane->bp[i])
      {
         compplane->bp[compbytes] = (unsigned char)((i & (255 << 16))
                                                    >> 16);
         compplane->bp[compbytes+1] = (unsigned char)((i & (255 << 8))
                                                      >> 8);
         compplane->bp[compbytes+2] = (unsigned char)(i & 255);
         compplane->bp[compbytes+3] = rawplane->bp[i];
         compbytes += 4;
      }

   }
   compplane->num_bytes = compbytes;

   return compplane;
}

//nonzero decoder
//num_bytes is uncompressed number of bytes in plane
PLANE *nzdecode(PLANE *compplane, unsigned int num_bytes)
{
   unsigned int i, byteposition;
   unsigned char j;
```

```
    //allocate memory for decompressed PLANE
    PLANE *rawplane = malloc(sizeof(PLANE));
    if (!rawplane)
       printf("Insufficient memory\n");

    //allocate memory for decompressed PLANE bit plane
    rawplane->bp = calloc(num_bytes,1);
    if (!(rawplane->bp))
       printf("Insufficient memory\n");

    //modify compression byte
    if (compplane->comptype != 1)
    {
       printf("Wrong compression type\n");
       return compplane;
    }

    else
       rawplane->comptype = 0;

    //insert nonzero pixels into array
    for (i = 0; i < compplane->num_bytes; i+=4)
    {
       byteposition = 0;
       for (j = 0; j < 3; j++)
          byteposition = (byteposition << 8) + compplane->bp[i+j];
       rawplane->bp[byteposition] = compplane->bp[i+3];
    }

    rawplane->num_bytes = num_bytes;

    return rawplane;
}
```

Appendix F: Sample Test Images

The following sequence of figures depict the set of test images used for the compression tests. For each test image, the actual image is reproduced, along with a contrast-heightened image to show detail. In addition, a histogram and image statistics are included for each test image.

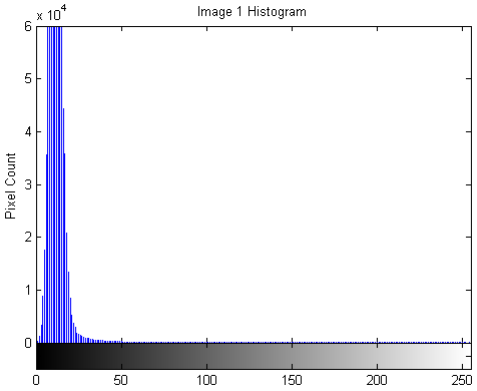


Image Entropy	Mean Pixel Value	Image Standard Deviation
3.91	12.28	9.70

Figure F-1: Test Image 1

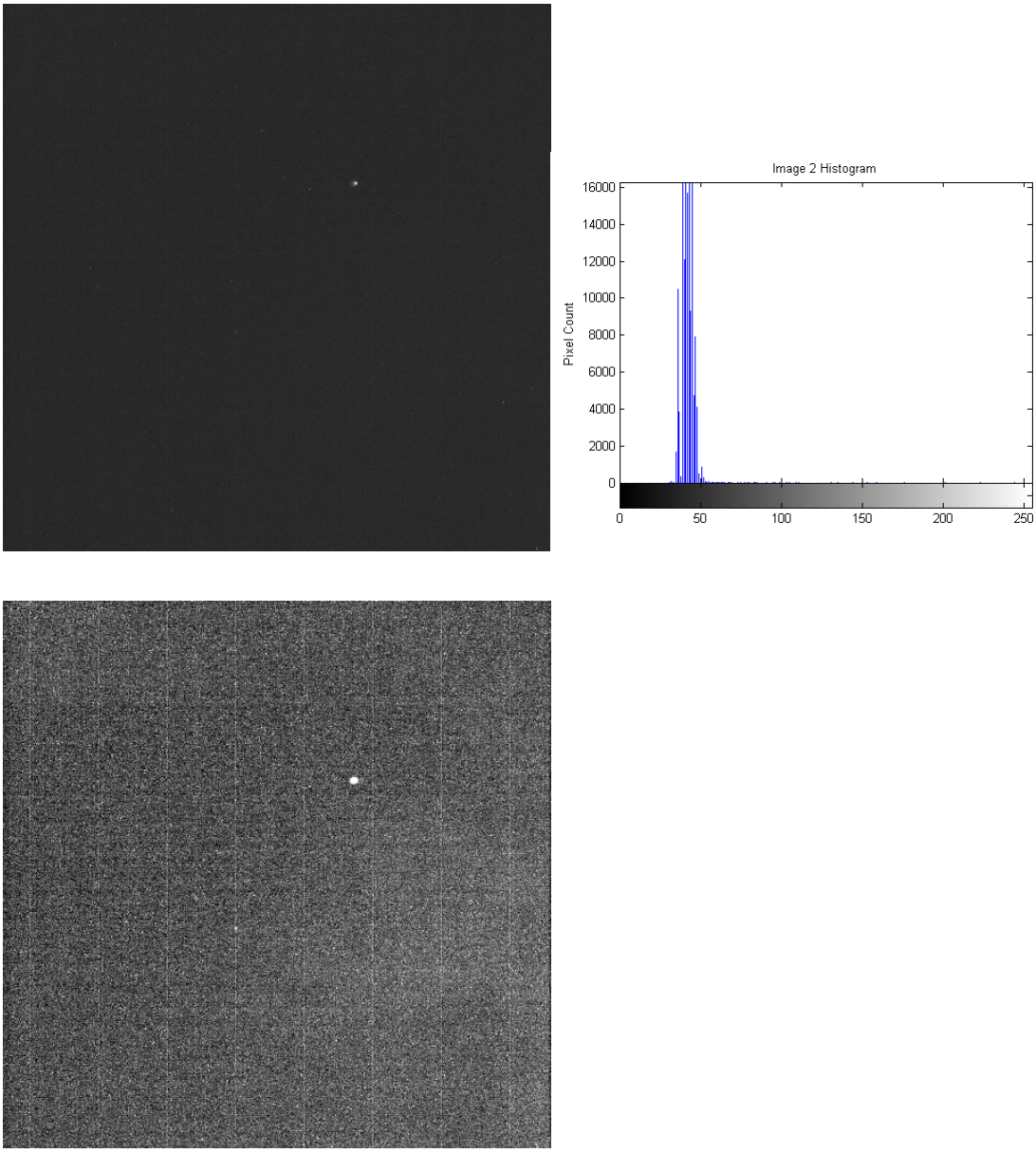
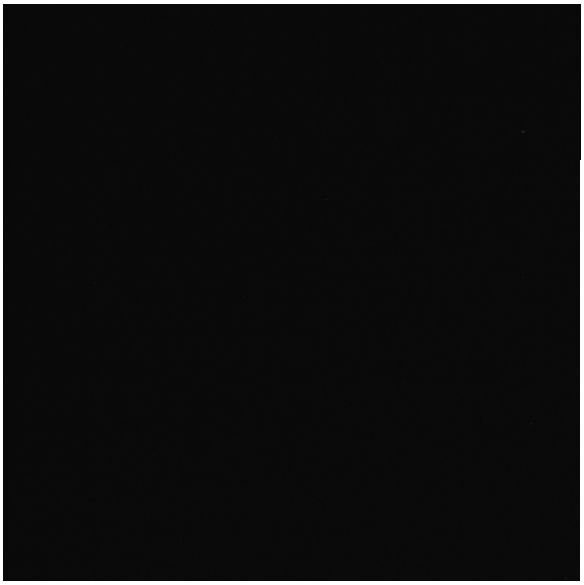
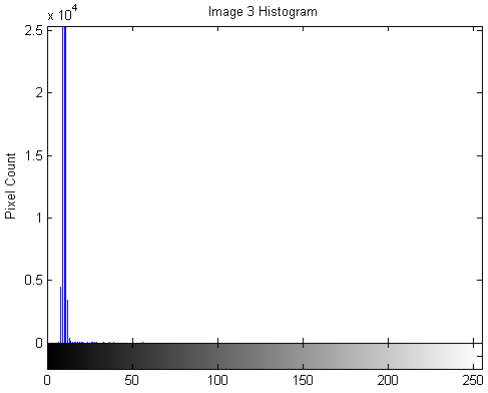


Figure F-2: Test Image 2

Image Entropy	Mean Pixel Value	Image Standard Deviation
3.10	41.60	2.94





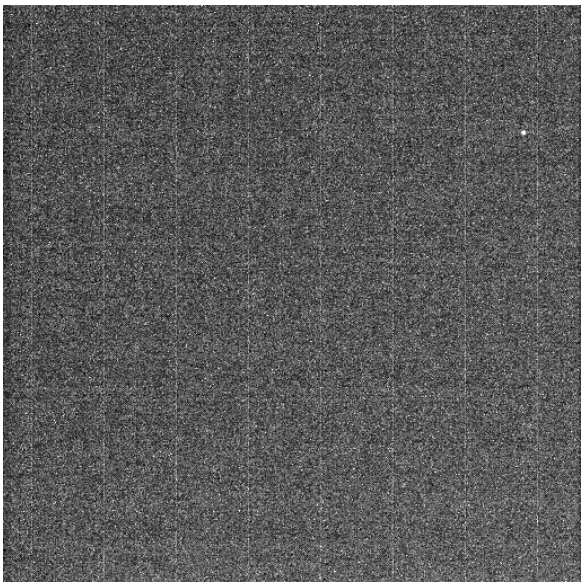
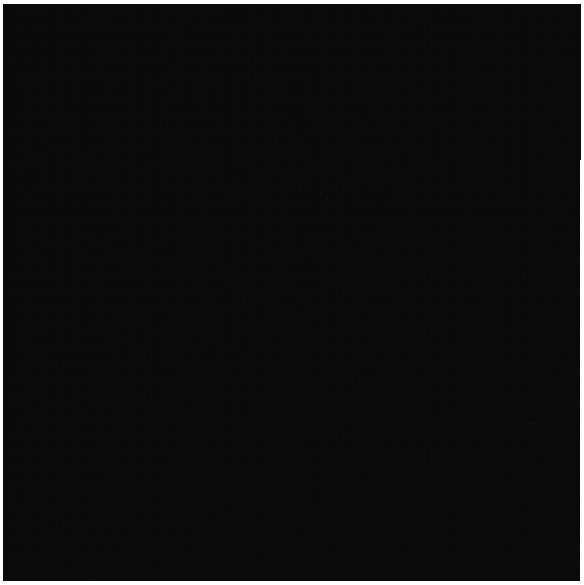
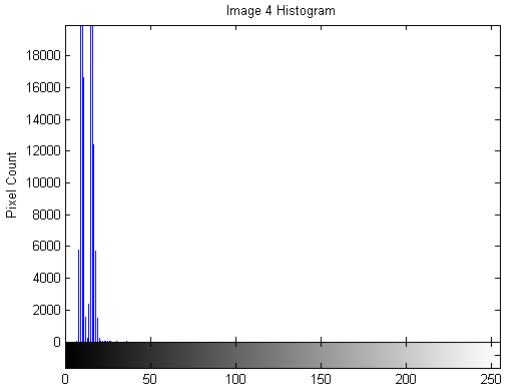


Figure F-3: Test Image 3

Image Entropy	Mean Pixel Value	Image Standard Deviation
1.60	9.74	0.77





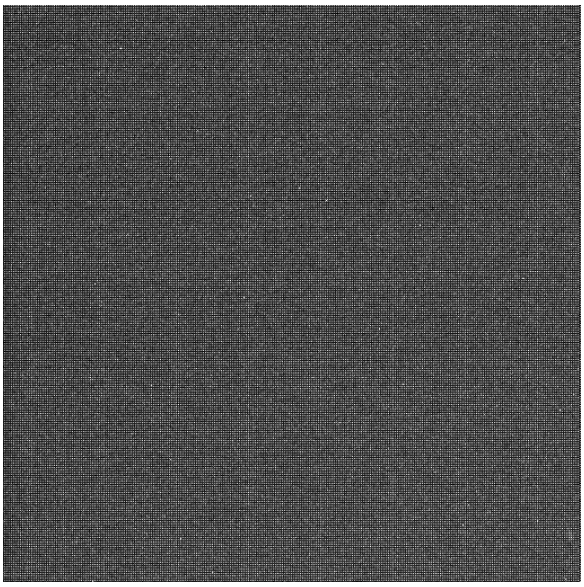


Figure F-4: Test Image 4

Image Entropy	Mean Pixel Value	Image Standard Deviation
2.52	11.23	2.93

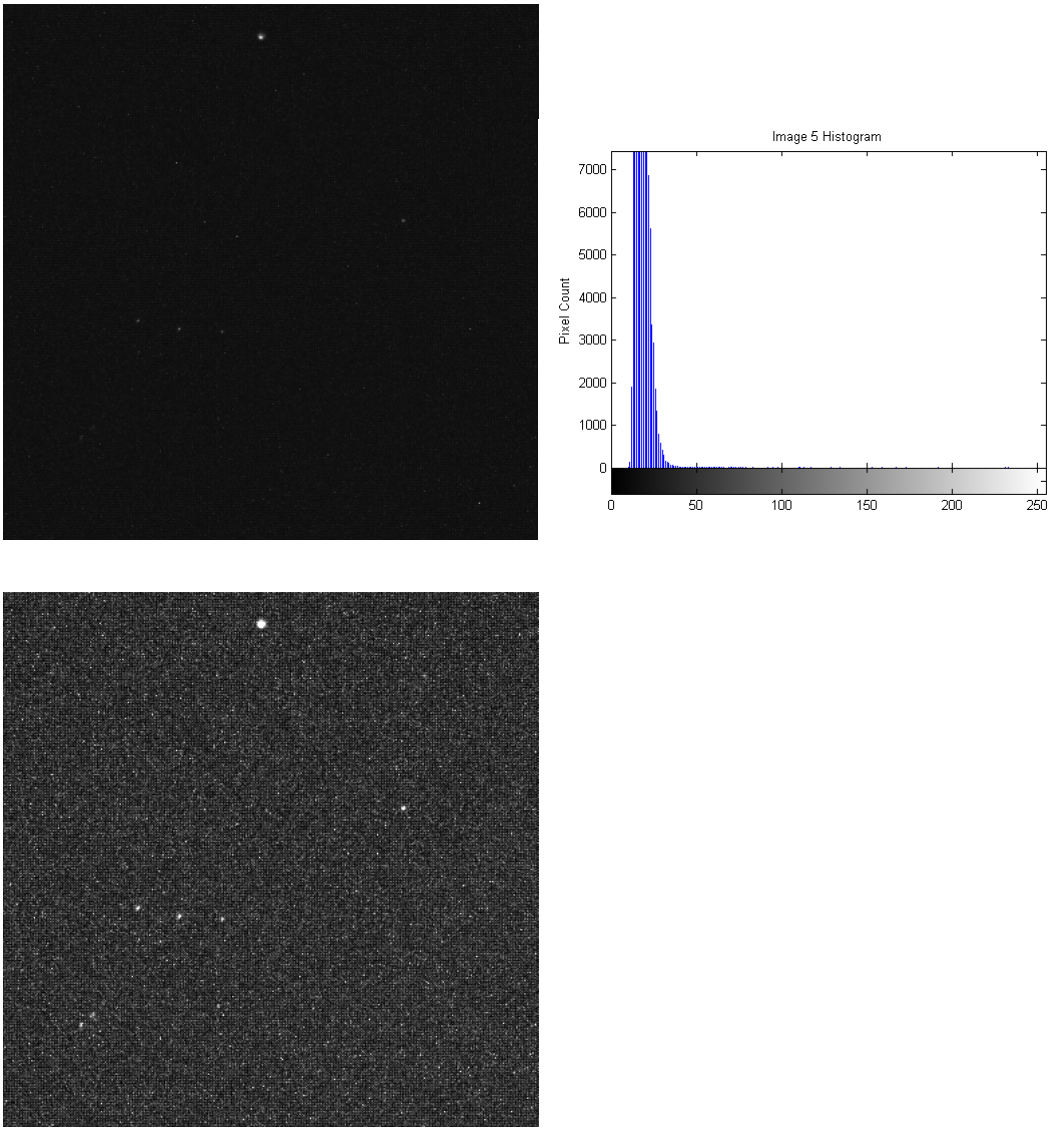


Figure F-5: Test Image 5

Image Entropy	Mean Pixel Value	Image Standard Deviation
3.77	17.86	4.10

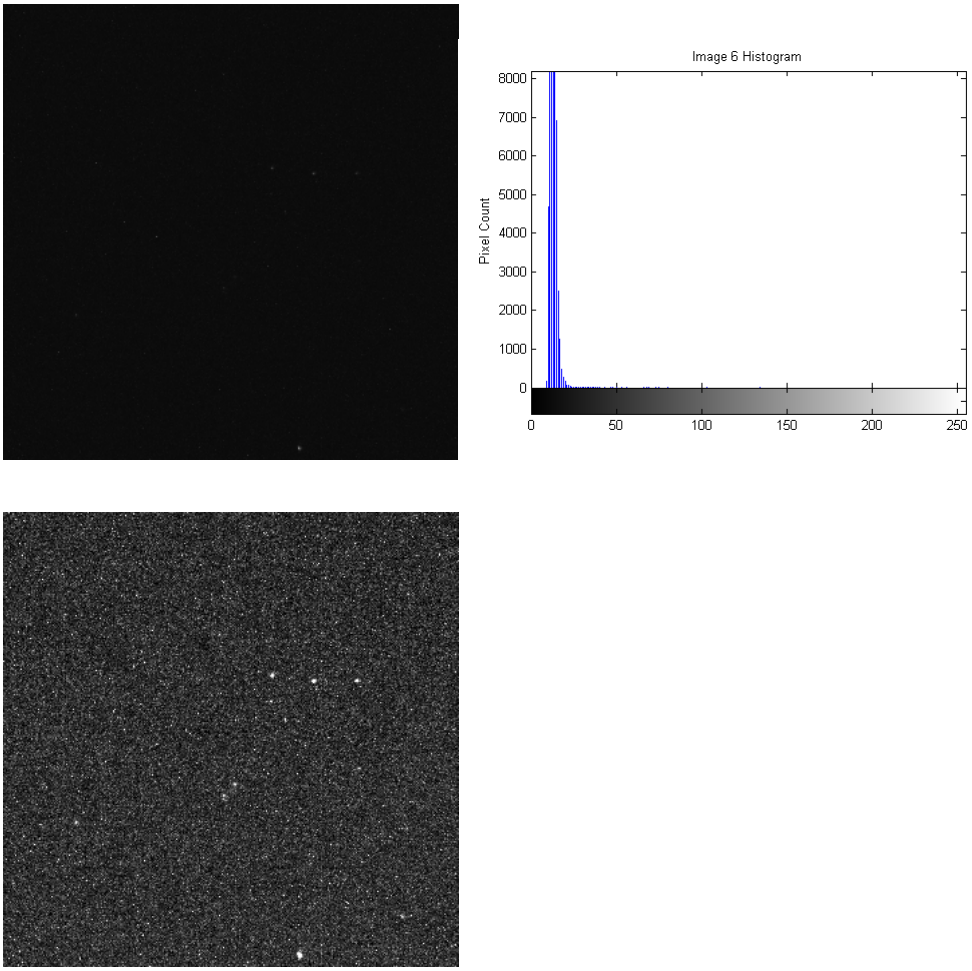
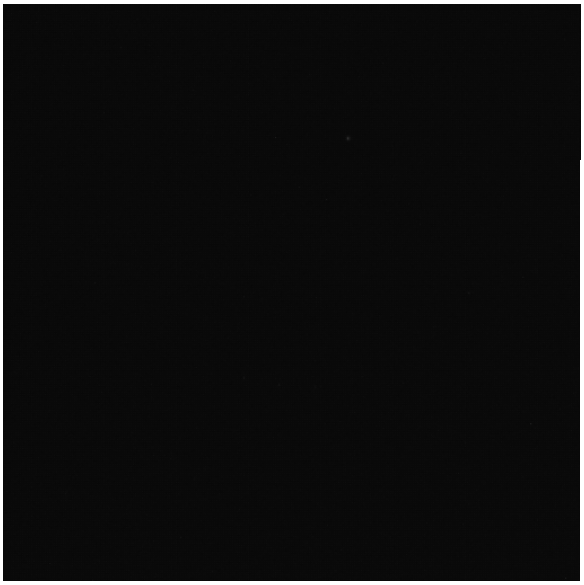
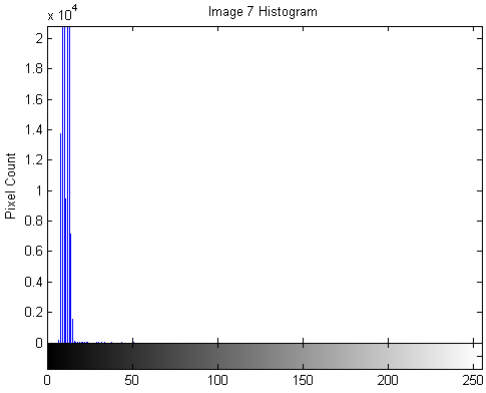


Figure F-6: Test Image 6

Image Entropy	Mean Pixel Value	Image Standard Deviation
2.57	12.60	1.77





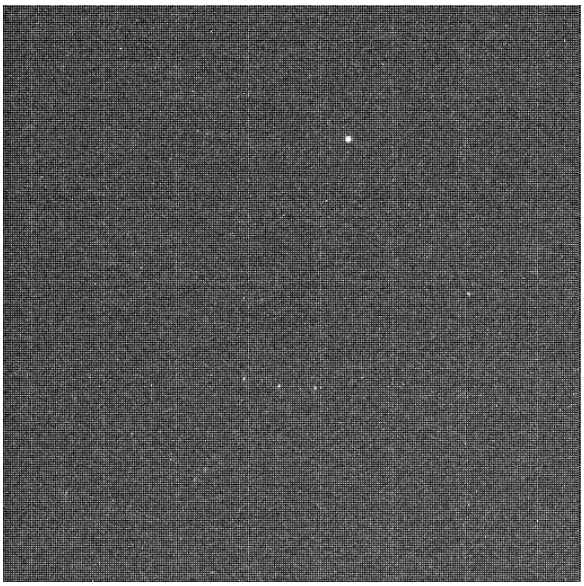
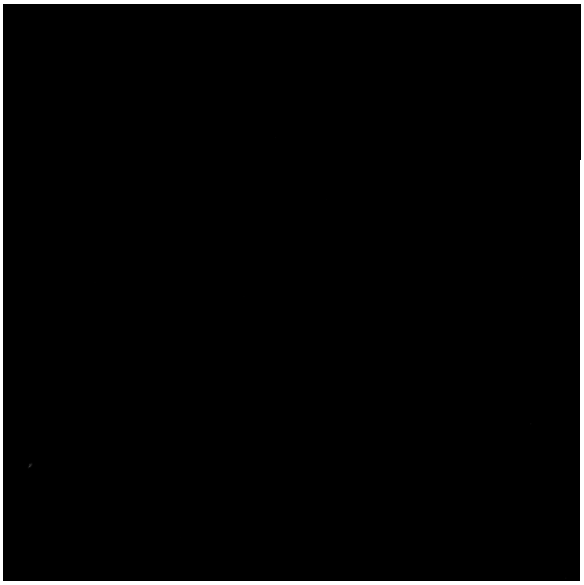


Figure F-7: Test Image 7

Image Entropy	Mean Pixel Value	Image Standard Deviation
2.32	10.21	1.64



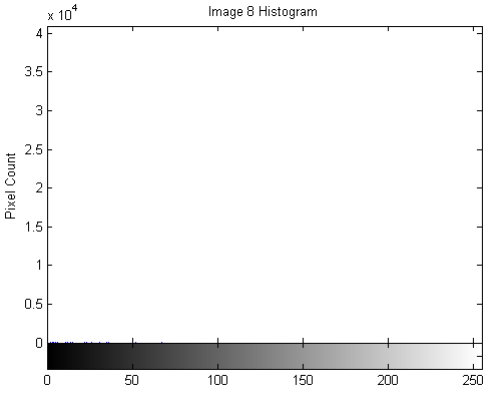


Figure F-8: Test Image 8

Image Entropy	Mean Pixel Value	Image Standard Deviation
0.002	0.002	0.23

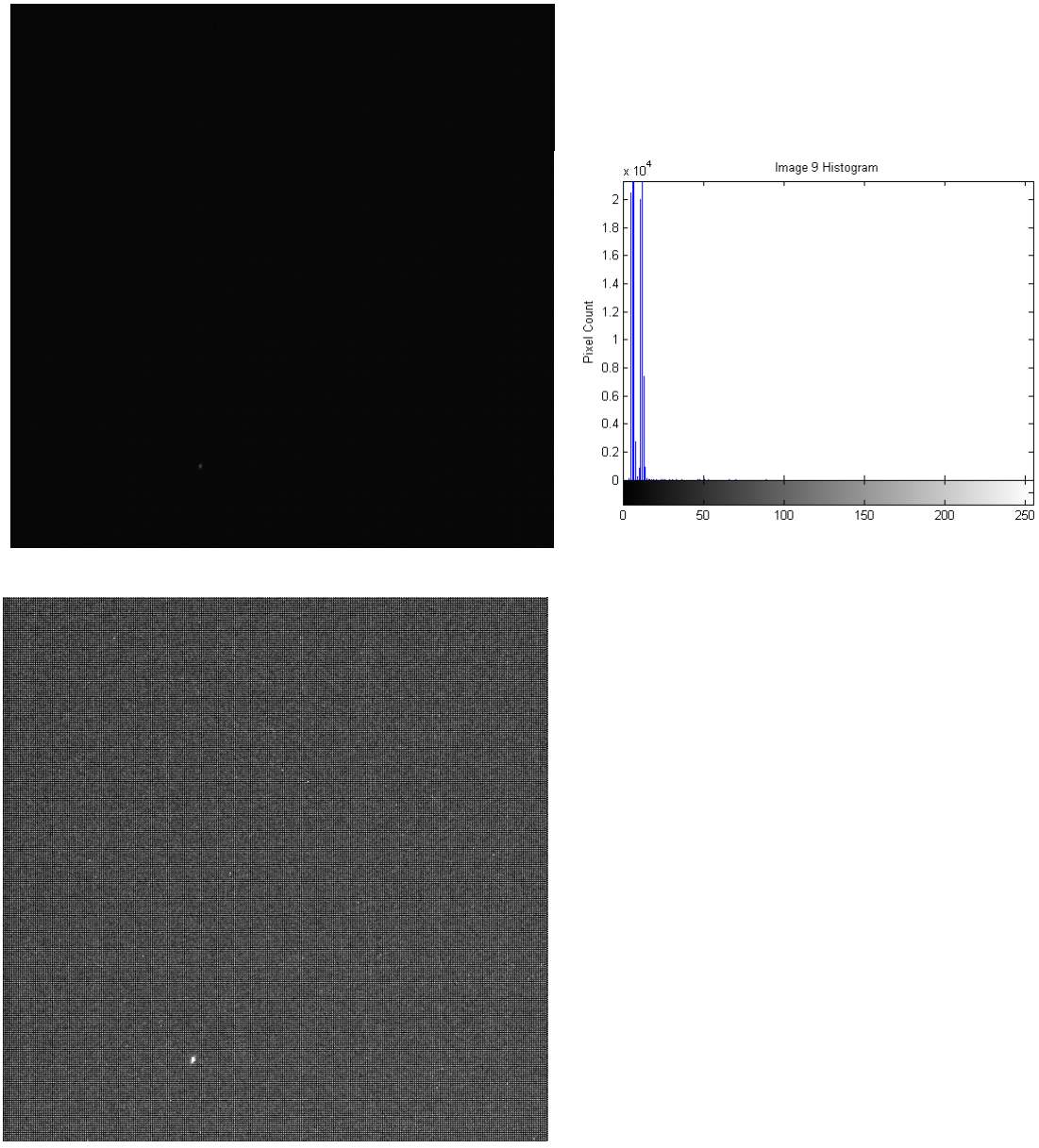
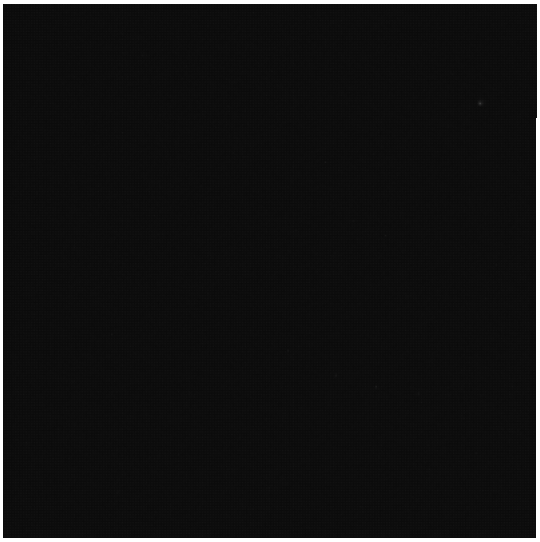
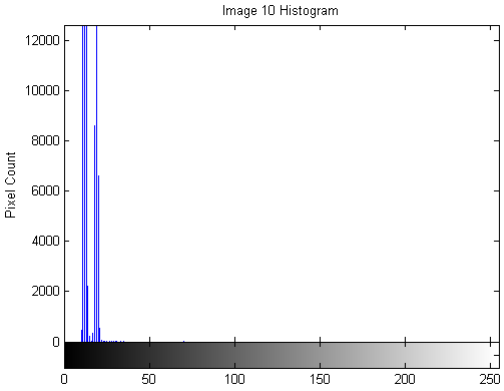


Figure F-9: Test Image 9

Image Entropy	Mean Pixel Value	Image Standard Deviation
2.26	7.63	2.53





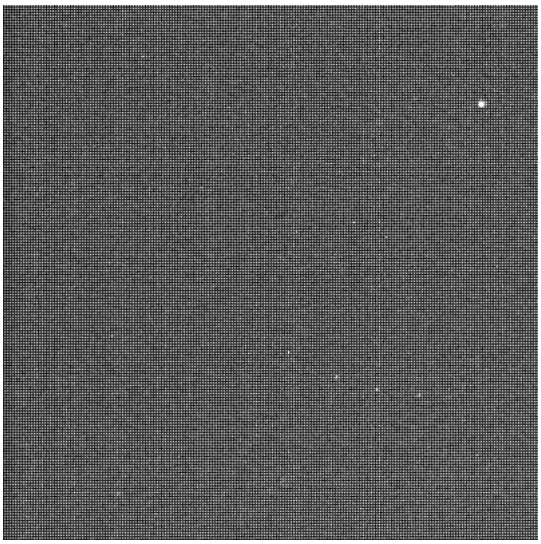
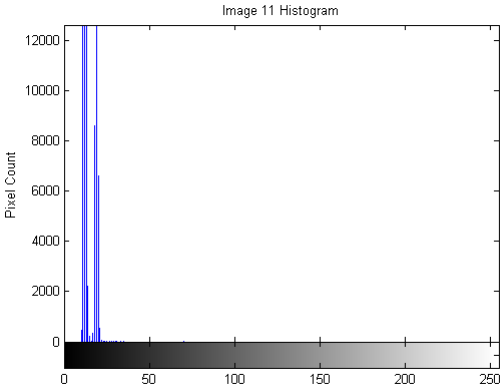


Figure F-10: Test Image 10

Image Entropy	Mean Pixel Value	Image Standard Deviation
2.36	13.78	3.09



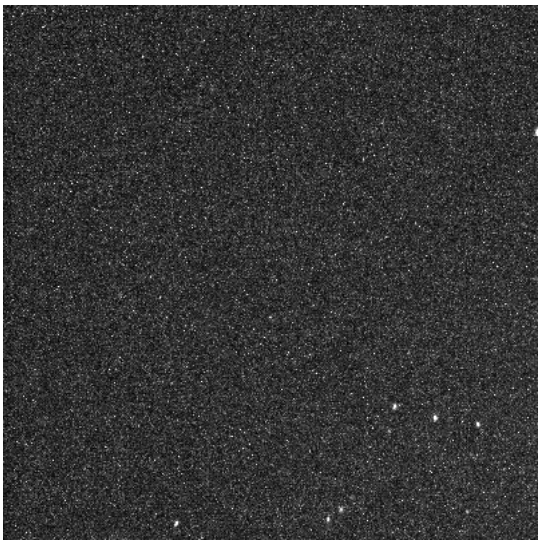


Figure F-11: Test Image 11

Image Entropy	Mean Pixel Value	Image Standard Deviation
2.36	13.78	3.09

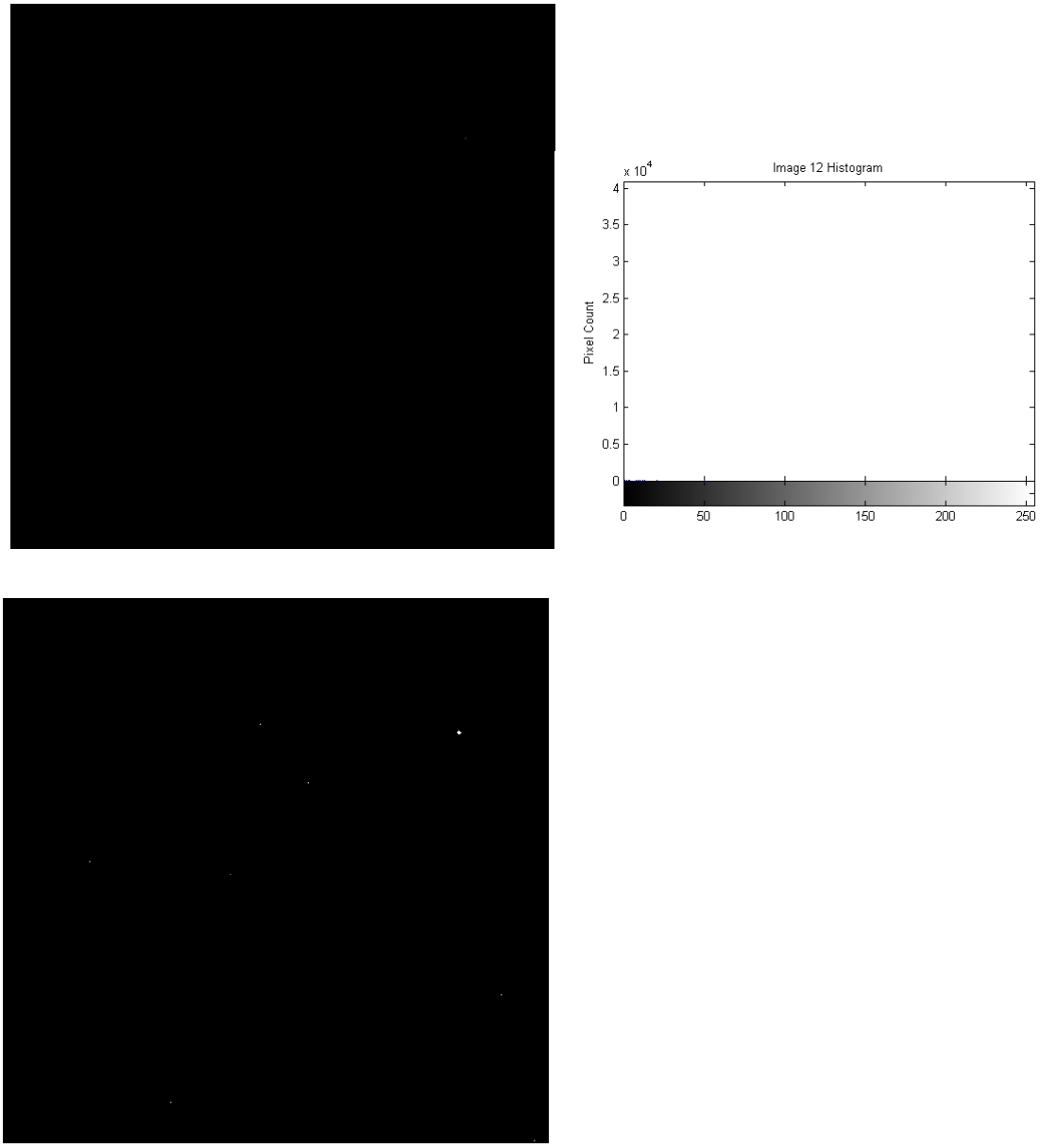


Figure F-12: Test Image 12

Image Entropy	Mean Pixel Value	Image Standard Deviation
0.001	0.001	0.12

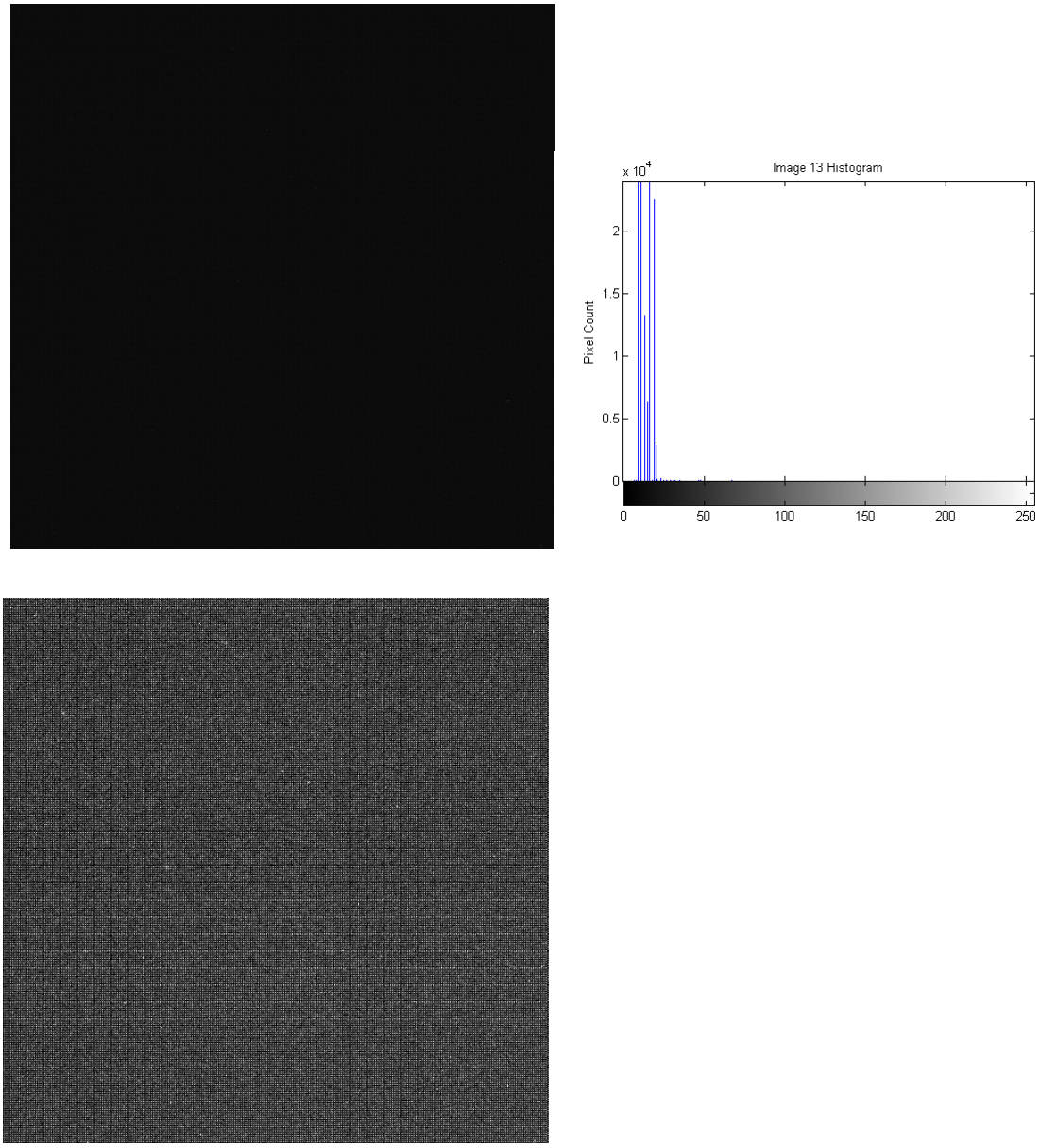


Figure F-13: Test Image 13

Image Entropy	Mean Pixel Value	Image Standard Deviation
2.02	12.33	3.05

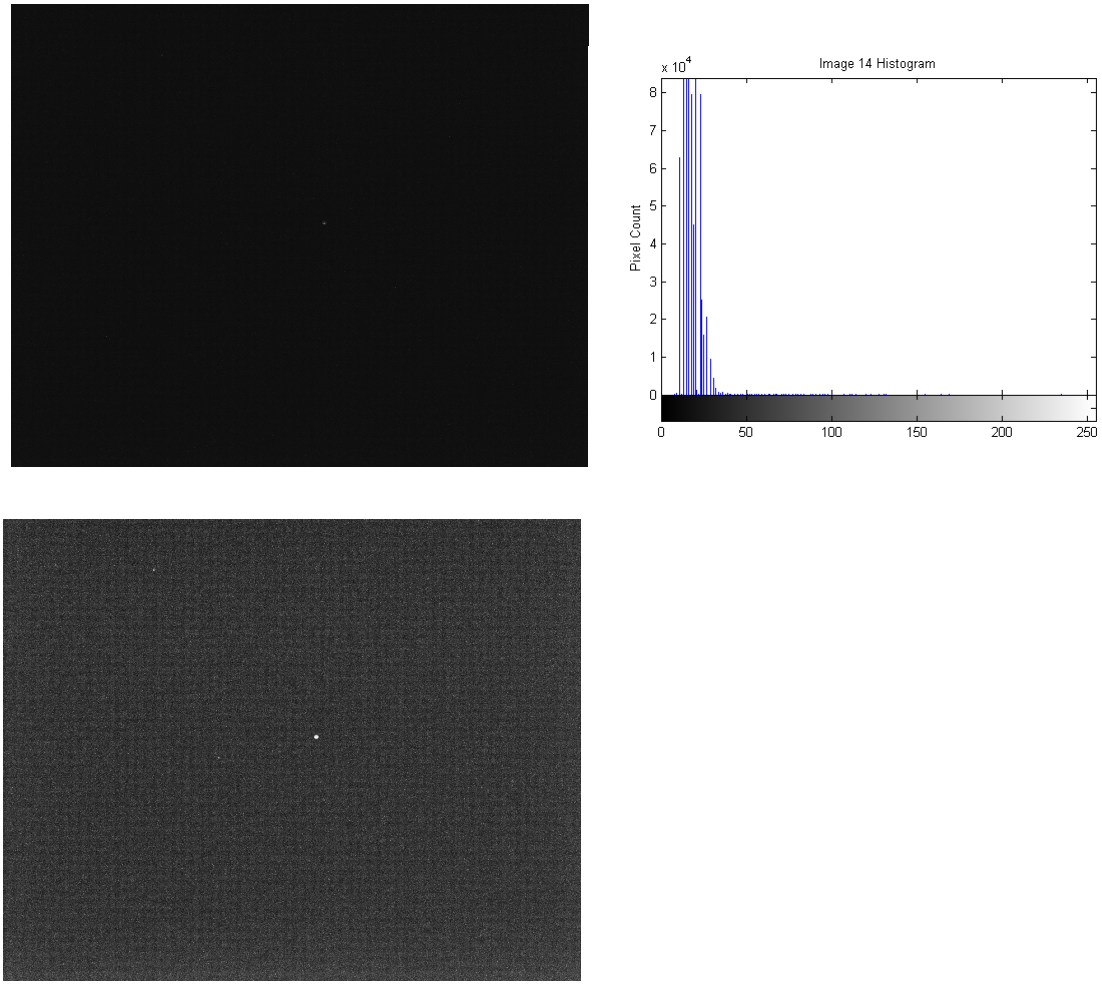


Figure F-14: Test Image 14

Image Entropy	Mean Pixel Value	Image Standard Deviation
2.99	16.41	4.04

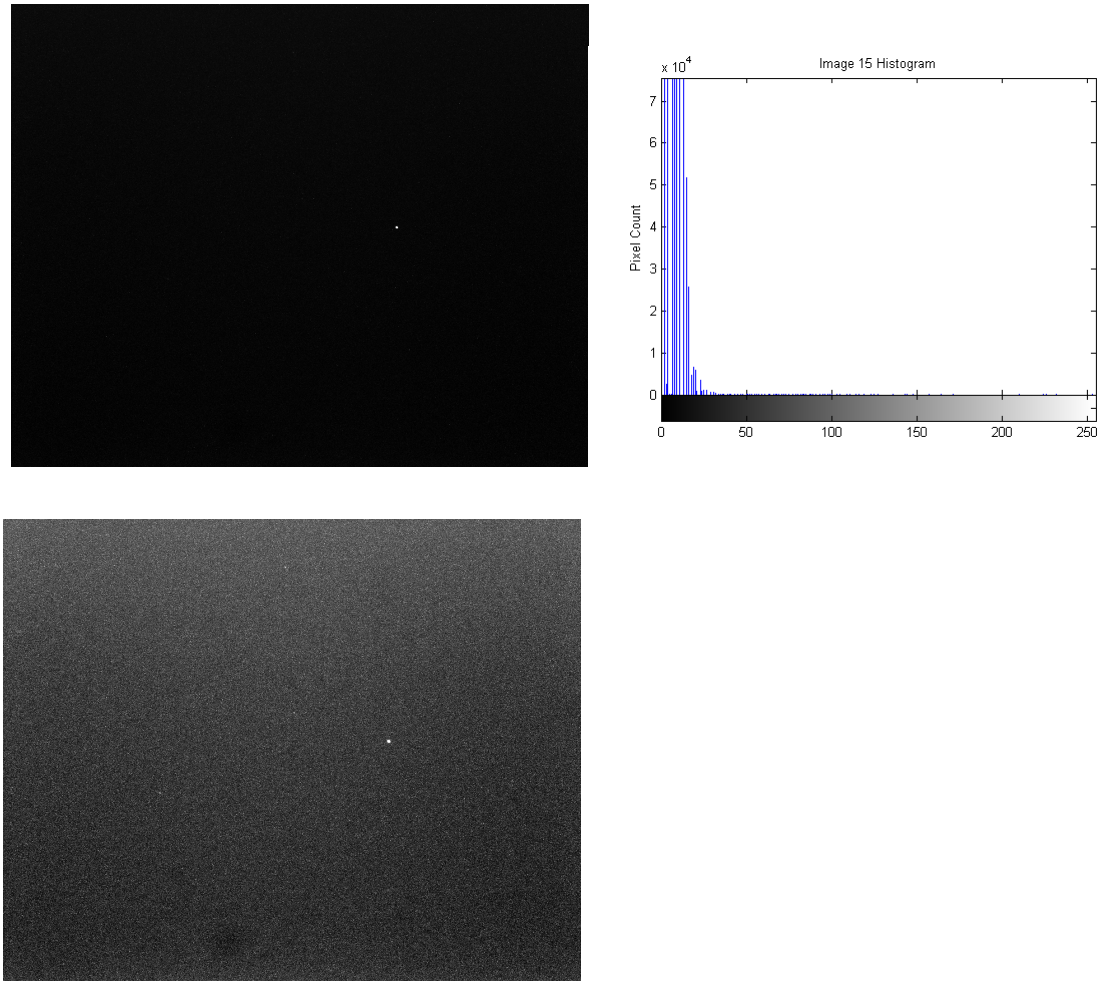


Figure F-15: Test Image 15

Image Entropy	Mean Pixel Value	Image Standard Deviation
3.18	7.68	4.59

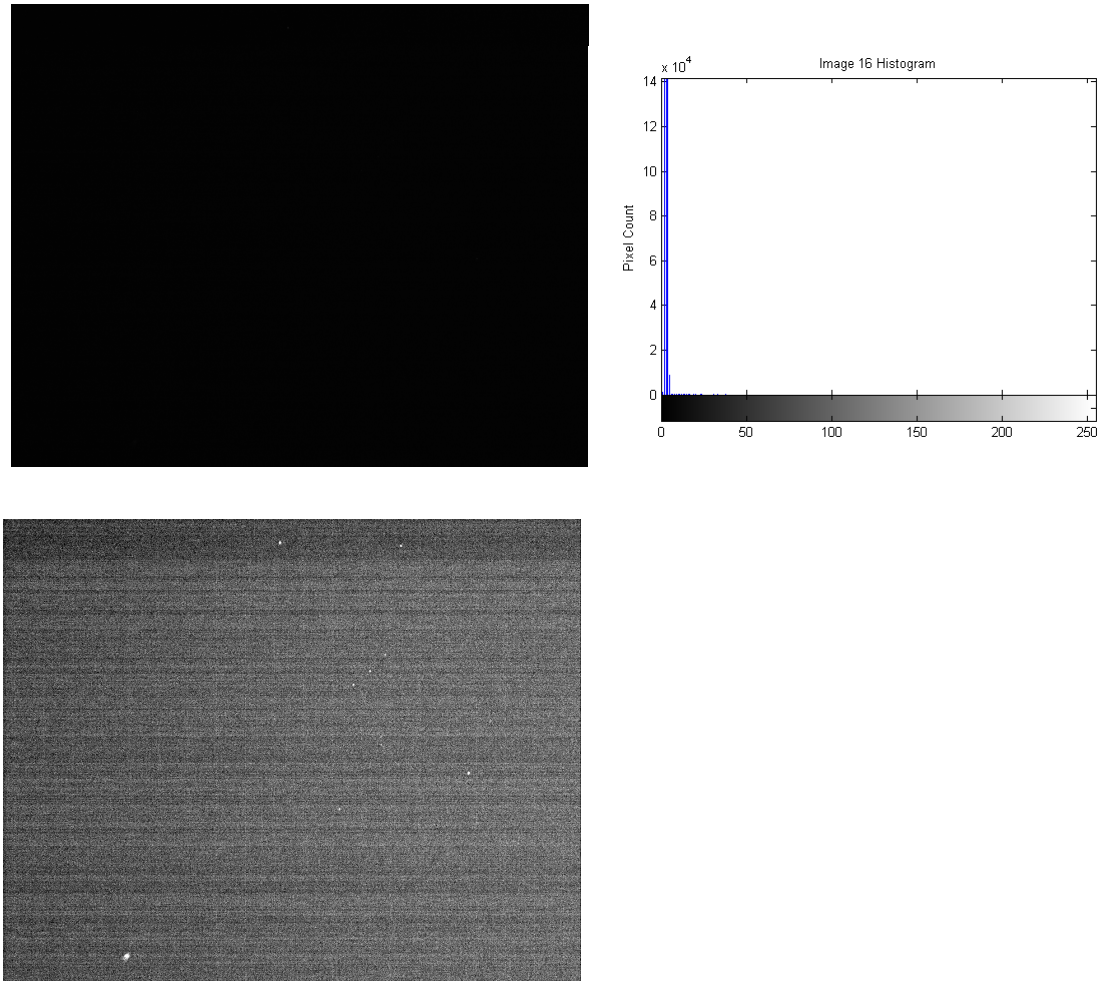


Figure F-16: Test Image 16

Image Entropy	Mean Pixel Value	Image Standard Deviation
1.32	3.14	0.61

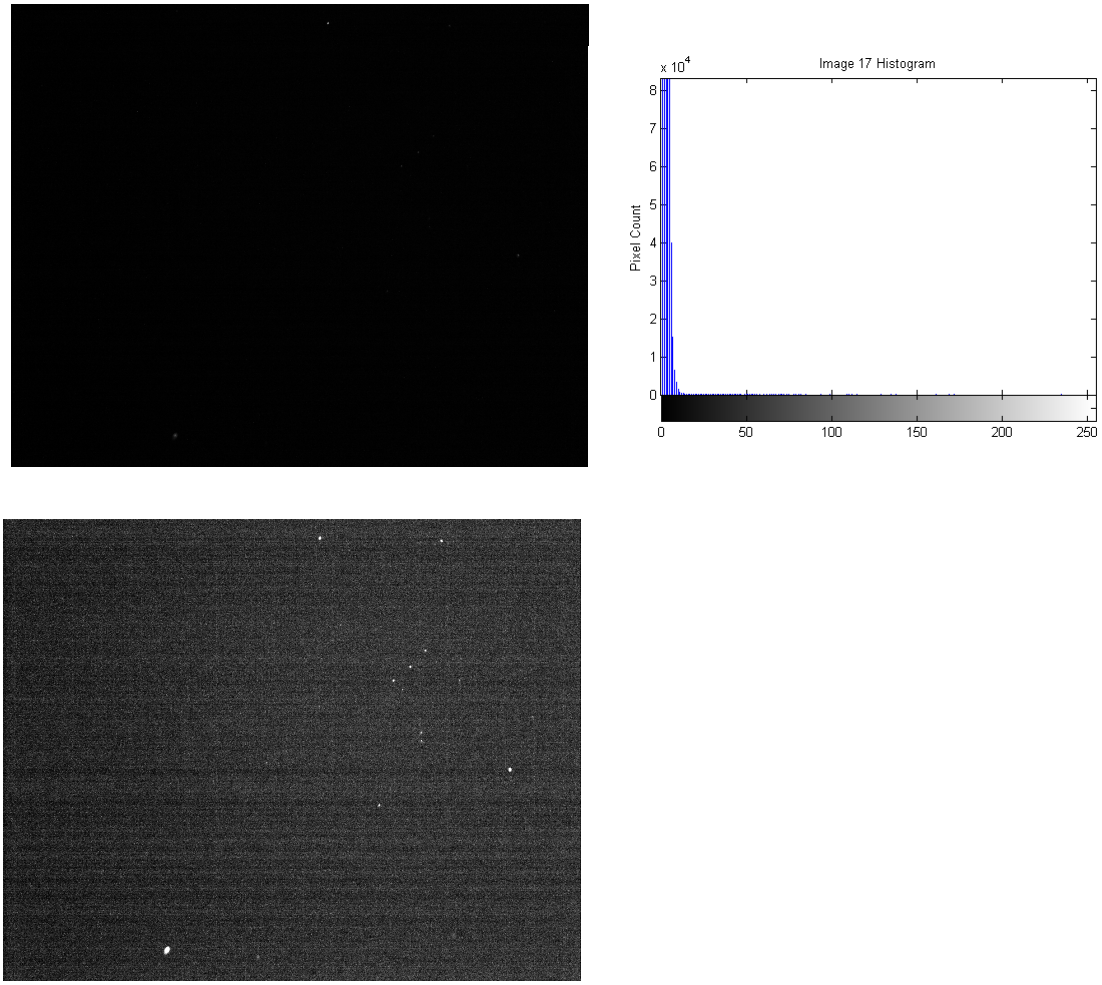


Figure F-17: Test Image 17

Image Entropy	Mean Pixel Value	Image Standard Deviation
2.77	2.32	2.01

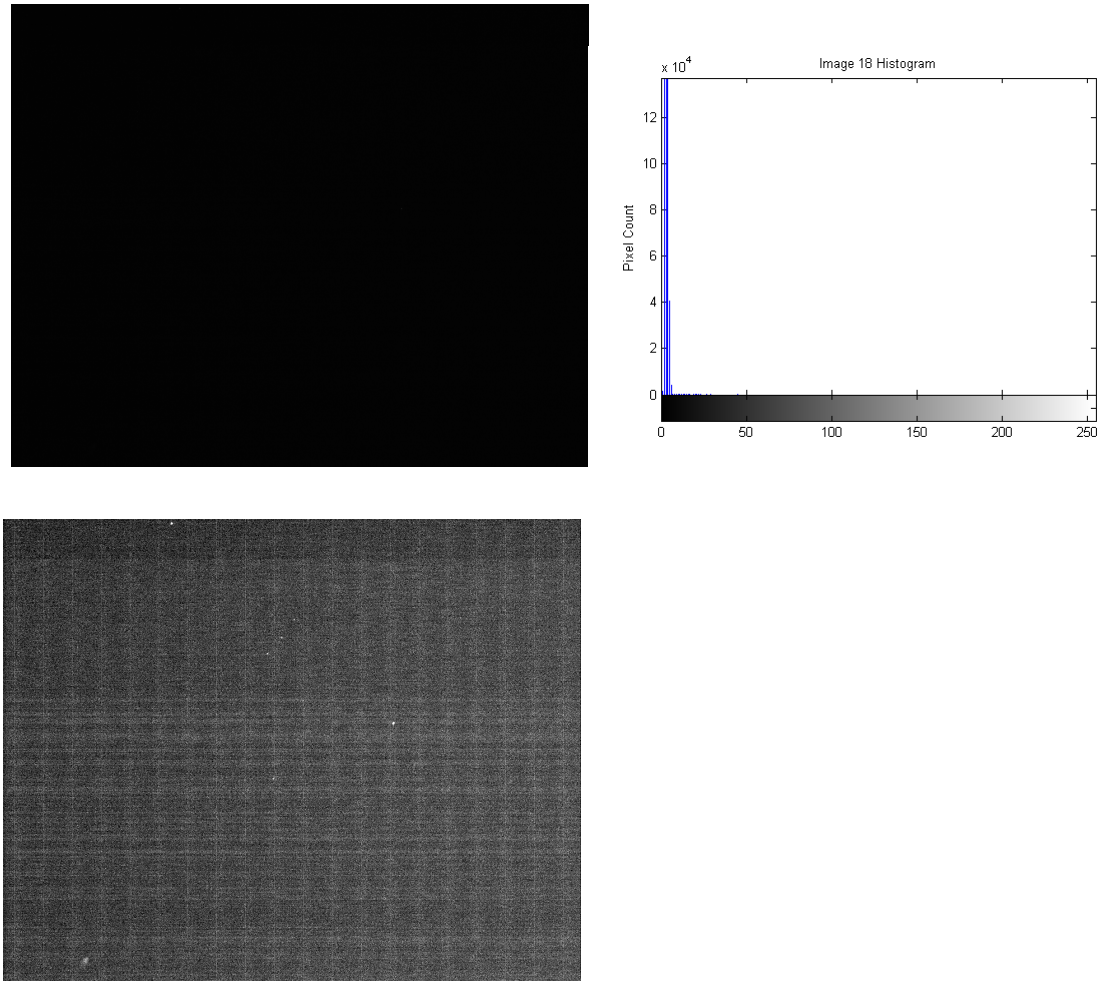


Figure F-18: Test Image 18

Image Entropy	Mean Pixel Value	Image Standard Deviation
1.48	3.15	0.70

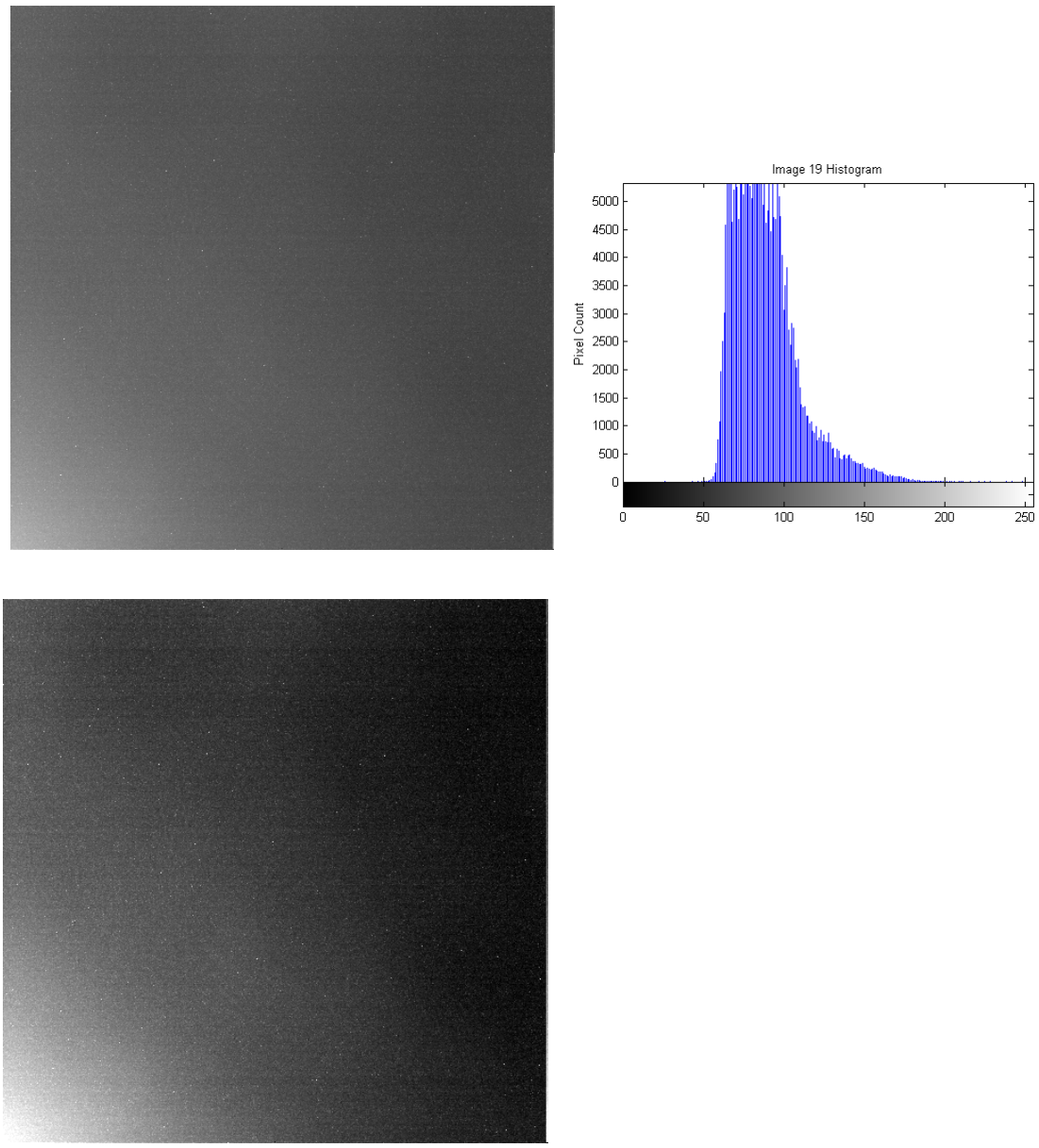


Figure F-19: Test Image 19

Image Entropy	Mean Pixel Value	Image Standard Deviation
6.15	88.86	20.24

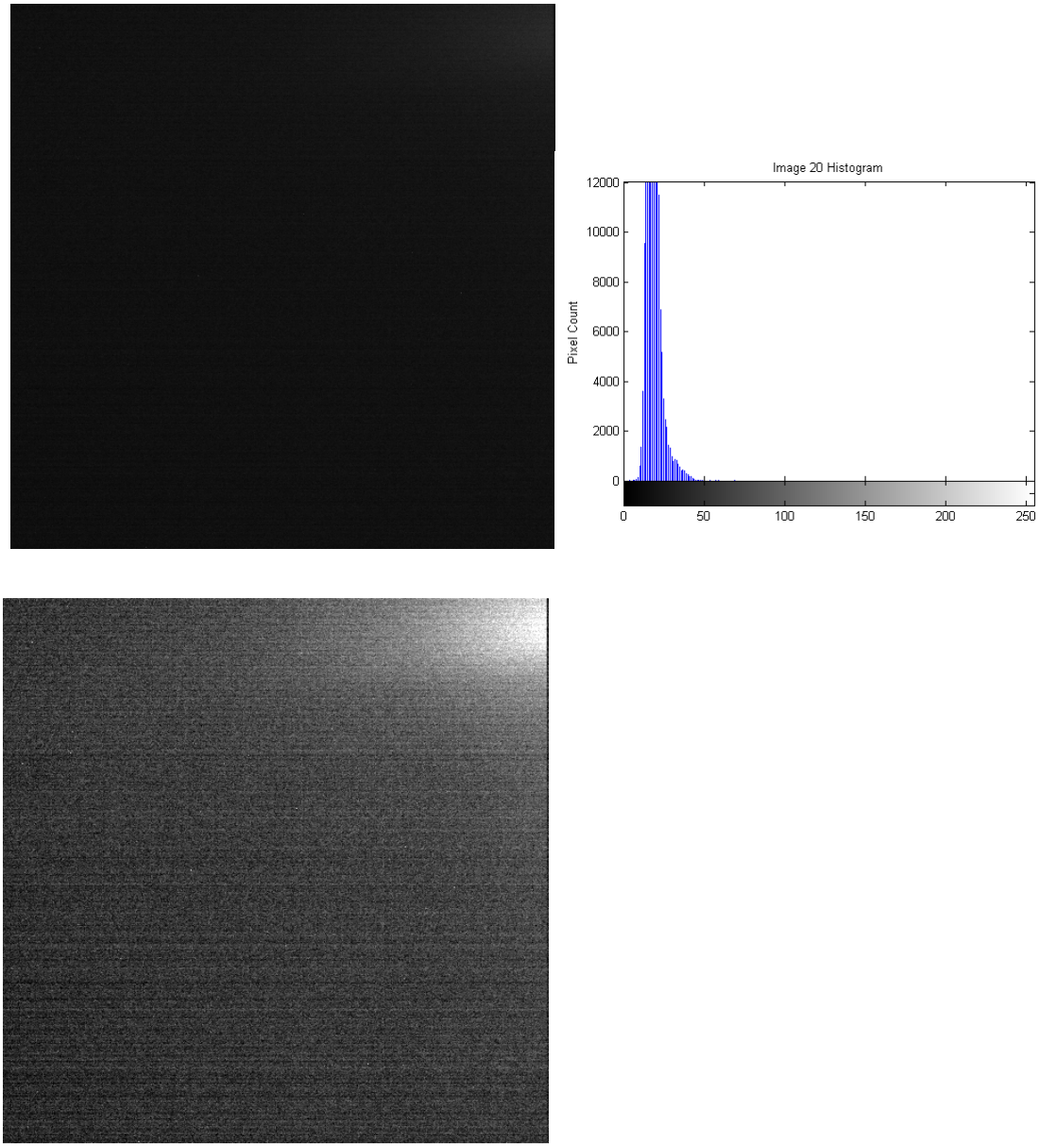
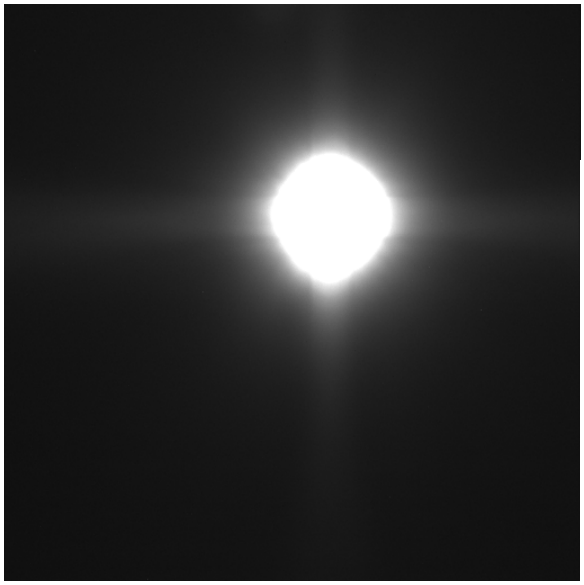
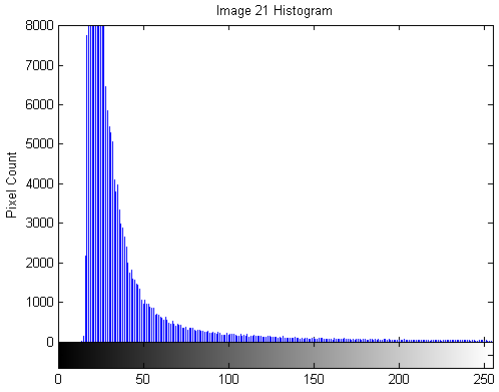


Figure F-20: Test Image 20

Image Entropy	Mean Pixel Value	Image Standard Deviation
3.88	18.50	4.21





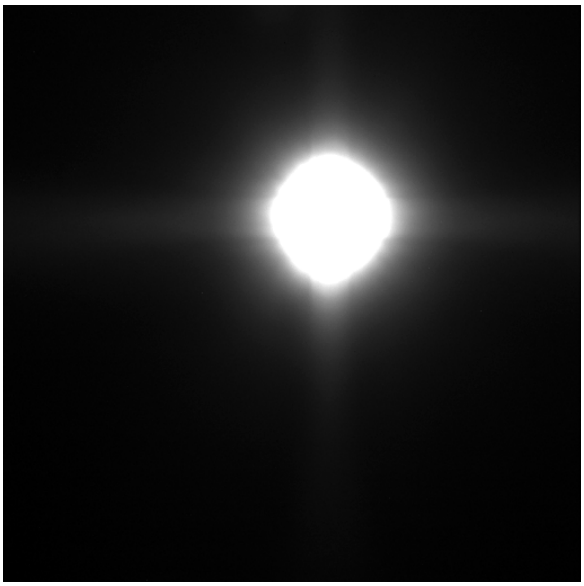
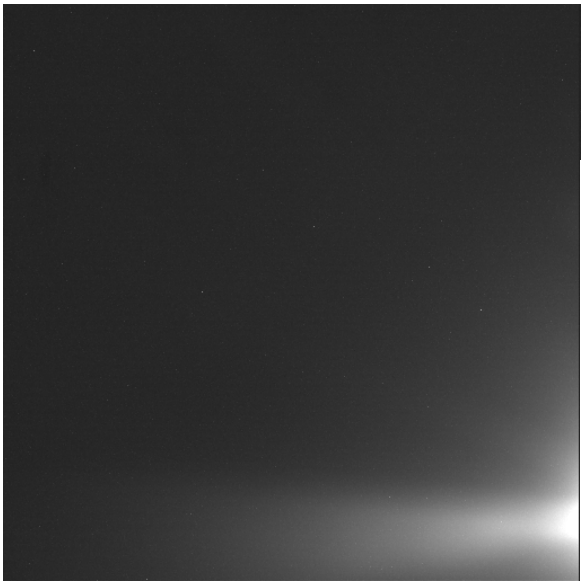


Figure F-21: Test Image 21

Image Entropy	Mean Pixel Value	Image Standard Deviation
5.48	42.11	49.34



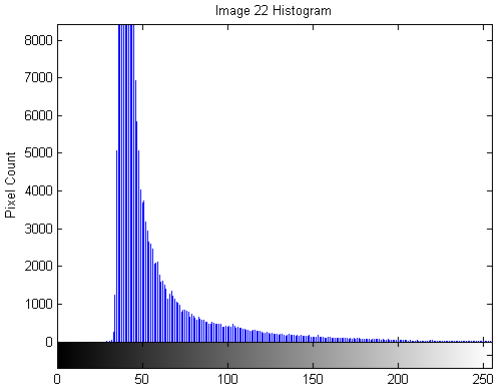


Figure F-22: Test Image 22

Image Entropy	Mean Pixel Value	Image Standard Deviation
5.43	53.00	27.97

Appendix G: Test Results

Complete compression test results are shown in the figures and table below.

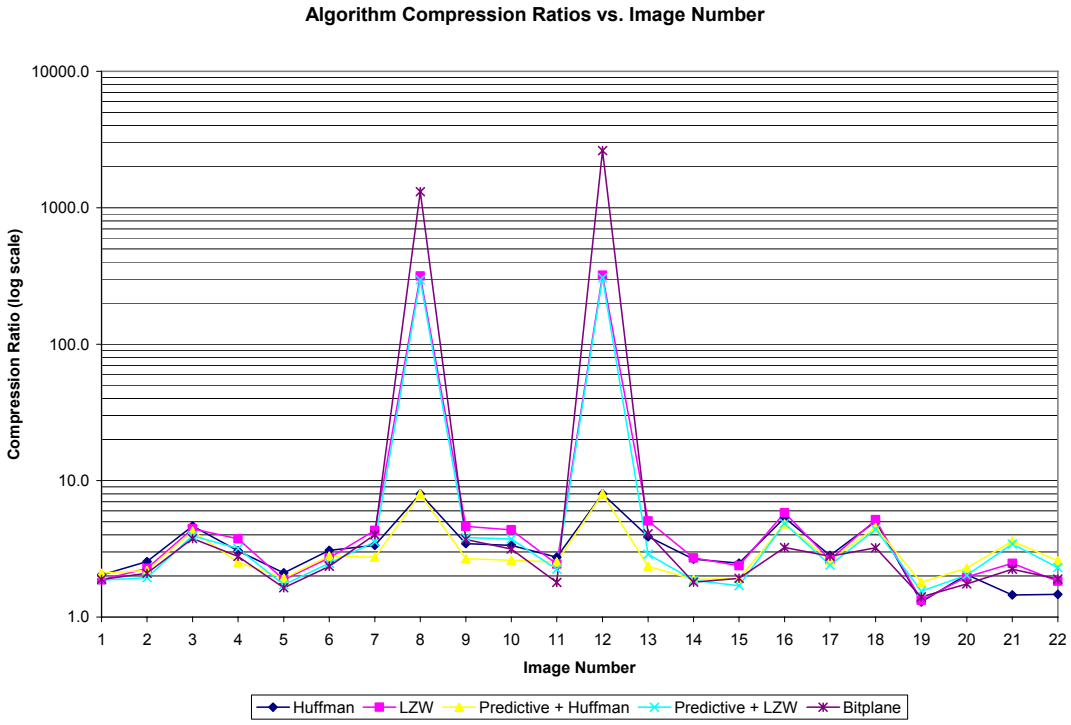


Figure G-1: Algorithm Compression Ratio vs. Image Number

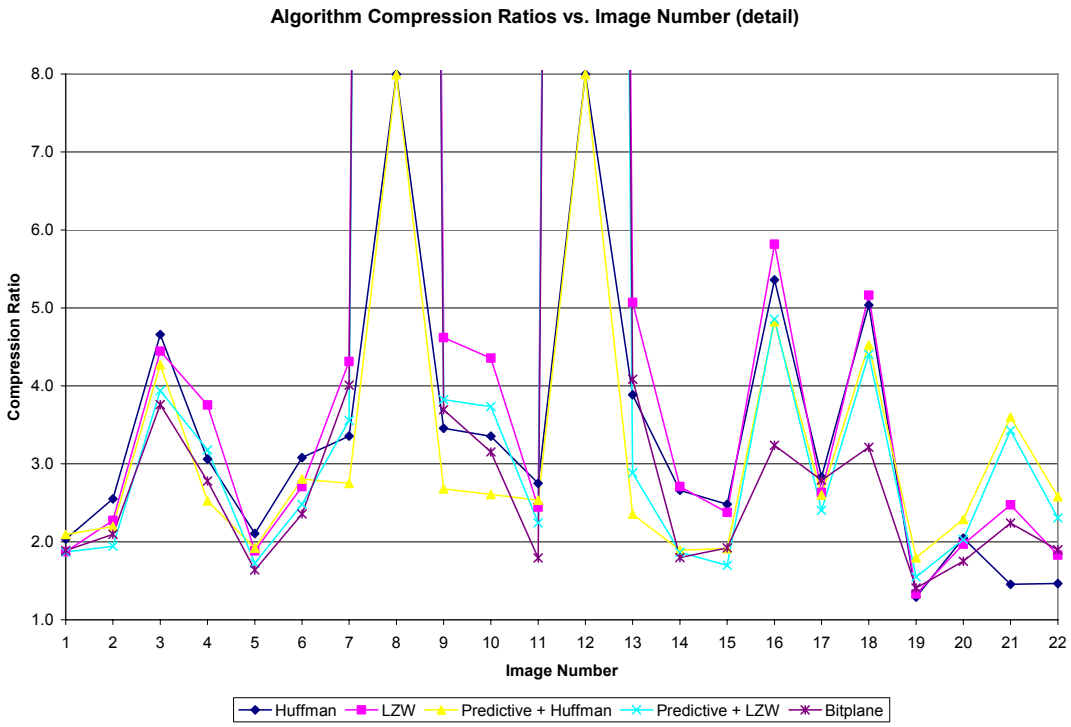


Figure G-2: Algorithm Compression Ratio vs. Image Number (detail)

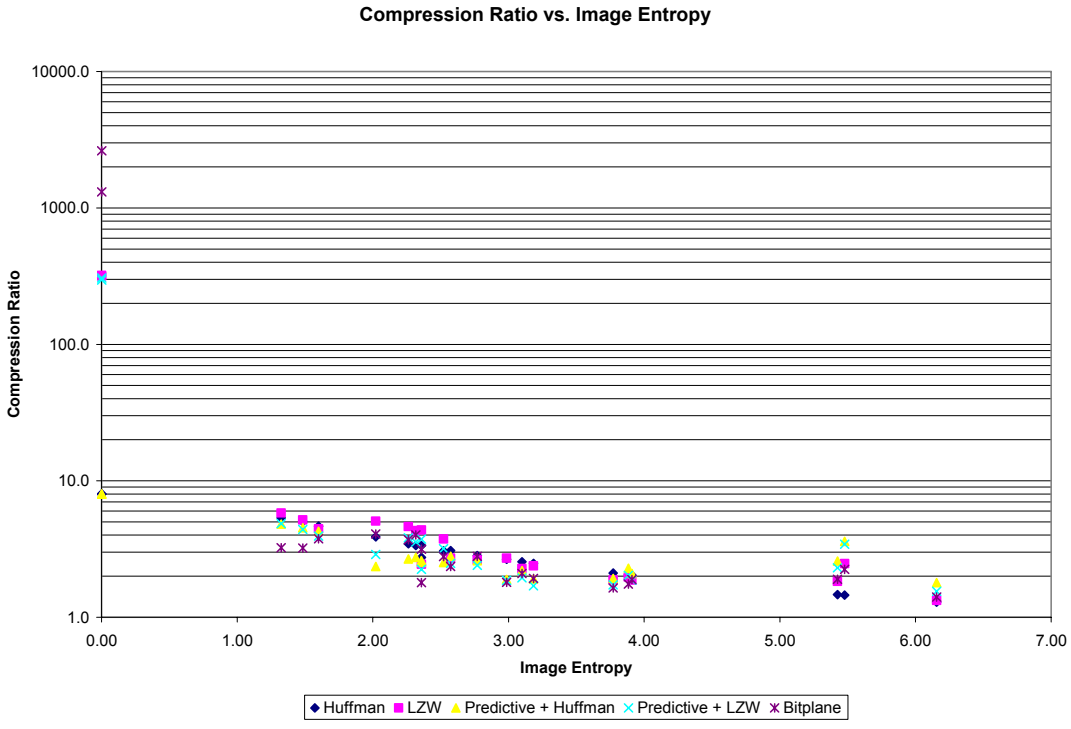


Figure G-3: Algorithm Compression Ratio vs. Image Entropy

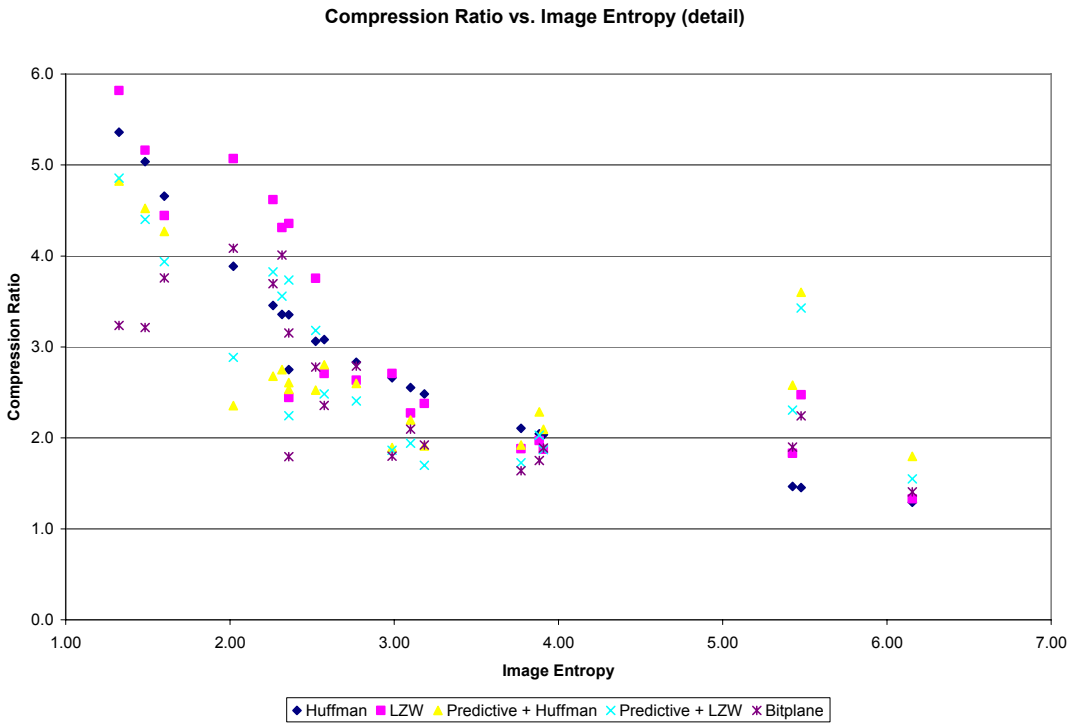


Figure G-4: Algorithm Compression Ratio vs. Image Entropy (detail)

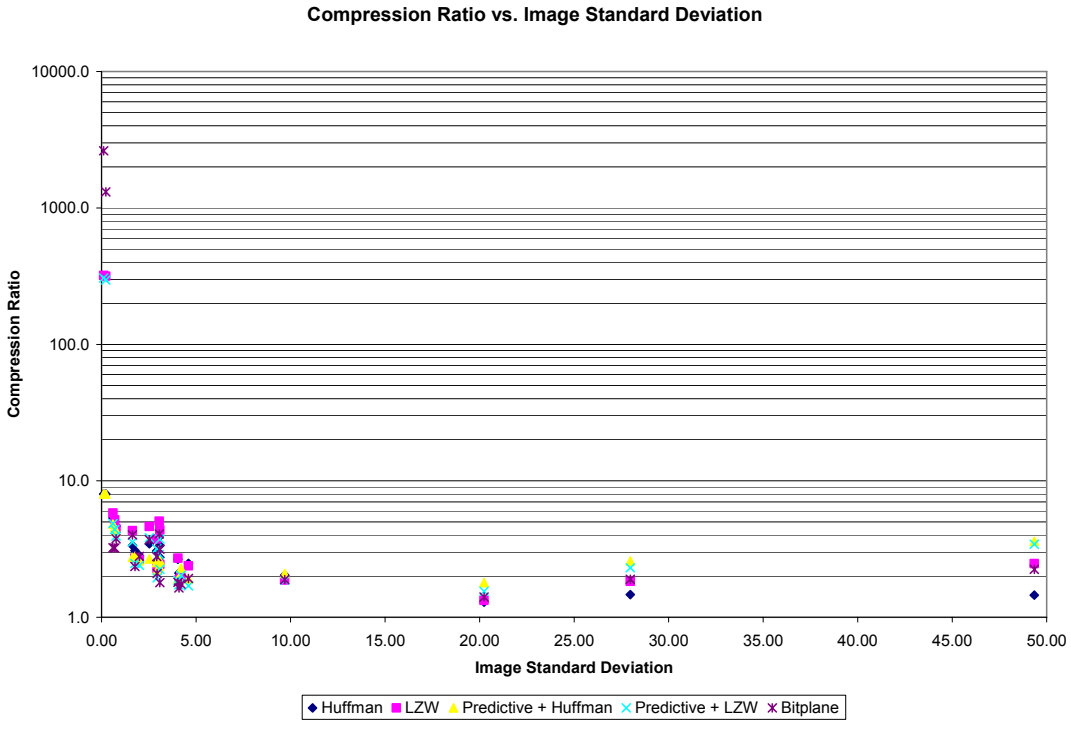


Figure G-5: Algorithm Compression Ratio vs. Image Standard Deviation

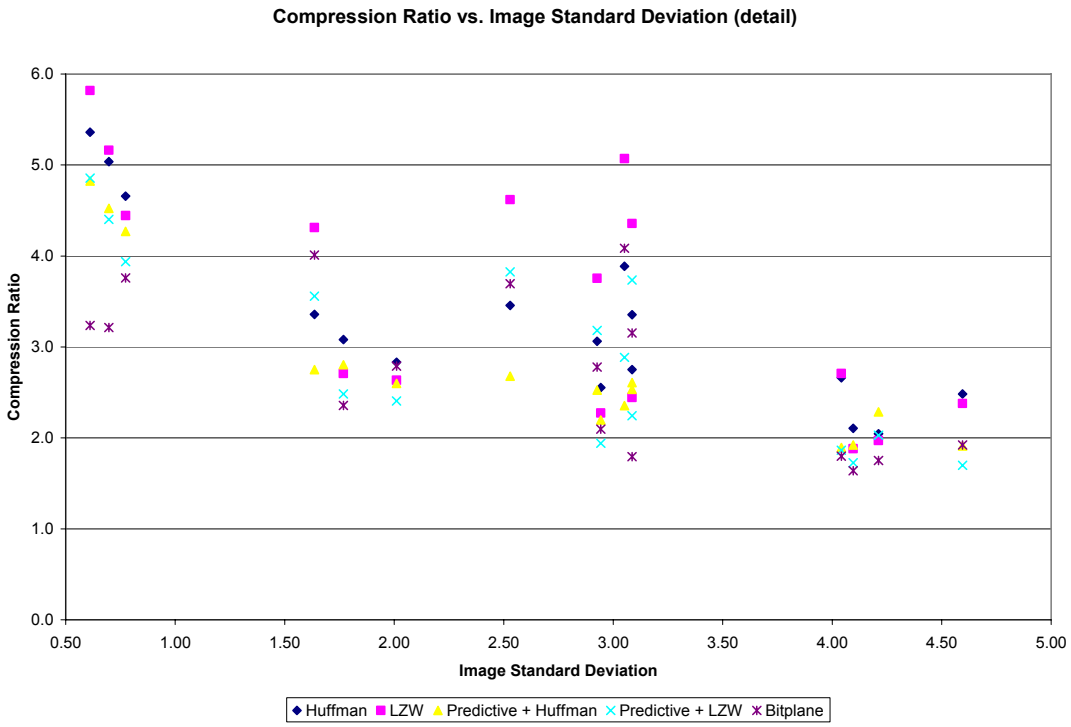


Figure G-6: Algorithm Compression Ratio vs. Image Standard Deviation (detail)

Image Number	Test Name	Test Result	Compressed Size	Original Size	Rows	Columns
1	Huffman	Match	644272	1310720	1024	1280
1	LZW	Match	699631	1310720	1024	1280
1	Predictive + Huffman	Match	625766	1310720	1024	1280
1	Predictive + LZW	Match	700824	1310720	1024	1280
1	Bit Plane 0 + Huffman	Match	20569	163840	1	163840
1	Bit Plane 0 + LZW	Match	805	163840	1	163840
1	Bit Plane 0 + Zero RLC	Match	1196	163840	1	163840
1	Bit Plane 1 + Huffman	Match	20809	163840	1	163840
1	Bit Plane 1 + LZW	Match	1227	163840	1	163840
1	Bit Plane 1 + Zero RLC	Match	2840	163840	1	163840
1	Bit Plane 2 + Huffman	Match	21985	163840	1	163840
1	Bit Plane 2 + LZW	Match	3273	163840	1	163840
1	Bit Plane 2 + Zero RLC	Match	9028	163840	1	163840
1	Bit Plane 3 + Huffman	Match	75622	163840	1	163840
1	Bit Plane 3 + LZW	Match	88637	163840	1	163840
1	Bit Plane 3 + Zero RLC	Match	321568	163840	1	163840
1	Bit Plane 4 + Huffman	Match	121955	163840	1	163840
1	Bit Plane 4 + LZW	Match	151370	163840	1	163840
1	Bit Plane 4 + Zero RLC	Match	653860	163840	1	163840
1	Bit Plane 5 + Huffman	Match	163840	163840	1	163840
1	Bit Plane 5 + LZW	Match	213960	163840	1	163840

Image Number	Test Name	Test Result	Compressed Size	Original Size	Rows	Columns
1	Bit Plane 5 + Zero RLC	Match	651964	163840	1	163840
1	Bit Plane 6 + Huffman	Match	163840	163840	1	163840
1	Bit Plane 6 + LZW	Match	213854	163840	1	163840
1	Bit Plane 6 + Zero RLC	Match	653156	163840	1	163840
1	Bit Plane 7 + Huffman	Match	163840	163840	1	163840
1	Bit Plane 7 + LZW	Match	214079	163840	1	163840
1	Bit Plane 7 + Zero RLC	Match	652844	163840	1	163840
2	Huffman	Match	102709	262144	512	512
2	LZW	Match	115275	262144	512	512
2	Predictive + Huffman	Match	119332	262144	512	512
2	Predictive + LZW	Match	135053	262144	512	512
2	Bit Plane 0 + Huffman	Match	4098	32768	1	32768
2	Bit Plane 0 + LZW	Match	237	32768	1	32768
2	Bit Plane 0 + Zero RLC	Match	20	32768	1	32768
2	Bit Plane 1 + Huffman	Match	4123	32768	1	32768
2	Bit Plane 1 + LZW	Match	346	32768	1	32768
2	Bit Plane 1 + Zero RLC	Match	252	32768	1	32768
2	Bit Plane 2 + Huffman	Match	4121	32768	1	32768
2	Bit Plane 2 + LZW	Match	340	32768	1	32768
2	Bit Plane 2 + Zero RLC	Match	131072	32768	1	32768
2	Bit Plane 3 + Huffman	Match	6616	32768	1	32768
2	Bit Plane 3 + LZW	Match	6399	32768	1	32768
2	Bit Plane 3 + Zero RLC	Match	21944	32768	1	32768
2	Bit Plane 4 + Huffman	Match	27945	32768	1	32768
2	Bit Plane 4 + LZW	Match	36971	32768	1	32768
2	Bit Plane 4 + Zero RLC	Match	131048	32768	1	32768
2	Bit Plane 5 + Huffman	Match	32347	32768	1	32768
2	Bit Plane 5 + LZW	Match	46331	32768	1	32768
2	Bit Plane 5 + Zero RLC	Match	129340	32768	1	32768
2	Bit Plane 6 + Huffman	Match	32767	32768	1	32768
2	Bit Plane 6 + LZW	Match	47958	32768	1	32768
2	Bit Plane 6 + Zero RLC	Match	130288	32768	1	32768
2	Bit Plane 7 + Huffman	Match	24933	32768	1	32768
2	Bit Plane 7 + LZW	Match	32090	32768	1	32768
2	Bit Plane 7 + Zero RLC	Match	131072	32768	1	32768
3	Huffman	Match	56280	262144	512	512
3	LZW	Match	58974	262144	512	512
3	Predictive + Huffman	Match	61388	262144	512	512
3	Predictive + LZW	Match	66583	262144	512	512
3	Bit Plane 0 + Huffman	Match	4096	32768	1	32768
3	Bit Plane 0 + LZW	Match	225	32768	1	32768
3	Bit Plane 0 + Zero RLC	Match	0	32768	1	32768
3	Bit Plane 1 + Huffman	Match	4096	32768	1	32768
3	Bit Plane 1 + LZW	Match	225	32768	1	32768
3	Bit Plane 1 + Zero RLC	Match	0	32768	1	32768
3	Bit Plane 2 + Huffman	Match	4097	32768	1	32768

Image Number	Test Name	Test Result	Compressed Size	Original Size	Rows	Columns
3	Bit Plane 2 + LZW	Match	233	32768	1	32768
3	Bit Plane 2 + Zero RLC	Match	12	32768	1	32768
3	Bit Plane 3 + Huffman	Match	4118	32768	1	32768
3	Bit Plane 3 + LZW	Match	336	32768	1	32768
3	Bit Plane 3 + Zero RLC	Match	220	32768	1	32768
3	Bit Plane 4 + Huffman	Match	4123	32768	1	32768
3	Bit Plane 4 + LZW	Match	360	32768	1	32768
3	Bit Plane 4 + Zero RLC	Match	131072	32768	1	32768
3	Bit Plane 5 + Huffman	Match	5697	32768	1	32768
3	Bit Plane 5 + LZW	Match	4690	32768	1	32768
3	Bit Plane 5 + Zero RLC	Match	14796	32768	1	32768
3	Bit Plane 6 + Huffman	Match	31711	32768	1	32768
3	Bit Plane 6 + LZW	Match	44051	32768	1	32768
3	Bit Plane 6 + Zero RLC	Match	130808	32768	1	32768
3	Bit Plane 7 + Huffman	Match	32753	32768	1	32768
3	Bit Plane 7 + LZW	Match	47754	32768	1	32768
3	Bit Plane 7 + Zero RLC	Match	130204	32768	1	32768
4	Huffman	Match	85614	262144	512	512
4	LZW	Match	69838	262144	512	512
4	Predictive + Huffman	Match	103863	262144	512	512
4	Predictive + LZW	Match	82410	262144	512	512
4	Bit Plane 0 + Huffman	Match	4096	32768	1	32768
4	Bit Plane 0 + LZW	Match	225	32768	1	32768
4	Bit Plane 0 + Zero RLC	Match	0	32768	1	32768
4	Bit Plane 1 + Huffman	Match	4096	32768	1	32768
4	Bit Plane 1 + LZW	Match	225	32768	1	32768
4	Bit Plane 1 + Zero RLC	Match	0	32768	1	32768
4	Bit Plane 2 + Huffman	Match	4097	32768	1	32768
4	Bit Plane 2 + LZW	Match	233	32768	1	32768
4	Bit Plane 2 + Zero RLC	Match	12	32768	1	32768
4	Bit Plane 3 + Huffman	Match	11541	32768	1	32768
4	Bit Plane 3 + LZW	Match	9532	32768	1	32768
4	Bit Plane 3 + Zero RLC	Match	64432	32768	1	32768
4	Bit Plane 4 + Huffman	Match	11565	32768	1	32768
4	Bit Plane 4 + LZW	Match	9610	32768	1	32768
4	Bit Plane 4 + Zero RLC	Match	131072	32768	1	32768
4	Bit Plane 5 + Huffman	Match	11821	32768	1	32768
4	Bit Plane 5 + LZW	Match	12131	32768	1	32768
4	Bit Plane 5 + Zero RLC	Match	55904	32768	1	32768
4	Bit Plane 6 + Huffman	Match	31048	32768	1	32768
4	Bit Plane 6 + LZW	Match	42671	32768	1	32768
4	Bit Plane 6 + Zero RLC	Match	129808	32768	1	32768
4	Bit Plane 7 + Huffman	Match	32299	32768	1	32768
4	Bit Plane 7 + LZW	Match	46170	32768	1	32768
4	Bit Plane 7 + Zero RLC	Match	130464	32768	1	32768
5	Huffman	Match	75973	160000	400	400

Image Number	Test Name	Test Result	Compressed Size	Original Size	Rows	Columns
5	LZW	Match	85005	160000	400	400
5	Predictive + Huffman	Match	83298	160000	400	400
5	Predictive + LZW	Match	92680	160000	400	400
5	Bit Plane 0 + Huffman	Match	2502	20000	1	20000
5	Bit Plane 0 + LZW	Match	181	20000	1	20000
5	Bit Plane 0 + Zero RLC	Match	28	20000	1	20000
5	Bit Plane 1 + Huffman	Match	2511	20000	1	20000
5	Bit Plane 1 + LZW	Match	214	20000	1	20000
5	Bit Plane 1 + Zero RLC	Match	96	20000	1	20000
5	Bit Plane 2 + Huffman	Match	2801	20000	1	20000
5	Bit Plane 2 + LZW	Match	1282	20000	1	20000
5	Bit Plane 2 + Zero RLC	Match	3176	20000	1	20000
5	Bit Plane 3 + Huffman	Match	17059	20000	1	20000
5	Bit Plane 3 + LZW	Match	22352	20000	1	20000
5	Bit Plane 3 + Zero RLC	Match	79936	20000	1	20000
5	Bit Plane 4 + Huffman	Match	19282	20000	1	20000
5	Bit Plane 4 + LZW	Match	27185	20000	1	20000
5	Bit Plane 4 + Zero RLC	Match	78424	20000	1	20000
5	Bit Plane 5 + Huffman	Match	19929	20000	1	20000
5	Bit Plane 5 + LZW	Match	29103	20000	1	20000
5	Bit Plane 5 + Zero RLC	Match	79812	20000	1	20000
5	Bit Plane 6 + Huffman	Match	19987	20000	1	20000
5	Bit Plane 6 + LZW	Match	29469	20000	1	20000
5	Bit Plane 6 + Zero RLC	Match	79800	20000	1	20000
5	Bit Plane 7 + Huffman	Match	19994	20000	1	20000
5	Bit Plane 7 + LZW	Match	29431	20000	1	20000
5	Bit Plane 7 + Zero RLC	Match	79792	20000	1	20000
6	Huffman	Match	37519	115600	340	340
6	LZW	Match	42684	115600	340	340
6	Predictive + Huffman	Match	41233	115600	340	340
6	Predictive + LZW	Match	46578	115600	340	340
6	Bit Plane 0 + Huffman	Match	1807	14450	1	14450
6	Bit Plane 0 + LZW	Match	142	14450	1	14450
6	Bit Plane 0 + Zero RLC	Match	4	14450	1	14450
6	Bit Plane 1 + Huffman	Match	1809	14450	1	14450
6	Bit Plane 1 + LZW	Match	154	14450	1	14450
6	Bit Plane 1 + Zero RLC	Match	28	14450	1	14450
6	Bit Plane 2 + Huffman	Match	1819	14450	1	14450
6	Bit Plane 2 + LZW	Match	198	14450	1	14450
6	Bit Plane 2 + Zero RLC	Match	124	14450	1	14450
6	Bit Plane 3 + Huffman	Match	3918	14450	1	14450
6	Bit Plane 3 + LZW	Match	4611	14450	1	14450
6	Bit Plane 3 + Zero RLC	Match	16856	14450	1	14450
6	Bit Plane 4 + Huffman	Match	3891	14450	1	14450
6	Bit Plane 4 + LZW	Match	4574	14450	1	14450
6	Bit Plane 4 + Zero RLC	Match	57800	14450	1	14450

Image Number	Test Name	Test Result	Compressed Size	Original Size	Rows	Columns
6	Bit Plane 5 + Huffman	Match	12395	14450	1	14450
6	Bit Plane 5 + LZW	Match	16820	14450	1	14450
6	Bit Plane 5 + Zero RLC	Match	57800	14450	1	14450
6	Bit Plane 6 + Huffman	Match	14252	14450	1	14450
6	Bit Plane 6 + LZW	Match	20803	14450	1	14450
6	Bit Plane 6 + Zero RLC	Match	56992	14450	1	14450
6	Bit Plane 7 + Huffman	Match	14447	14450	1	14450
6	Bit Plane 7 + LZW	Match	21242	14450	1	14450
6	Bit Plane 7 + Zero RLC	Match	57560	14450	1	14450
7	Huffman	Match	78088	262144	512	512
7	LZW	Match	60787	262144	512	512
7	Predictive + Huffman	Match	95326	262144	512	512
7	Predictive + LZW	Match	73678	262144	512	512
7	Bit Plane 0 + Huffman	Match	4096	32768	1	32768
7	Bit Plane 0 + LZW	Match	225	32768	1	32768
7	Bit Plane 0 + Zero RLC	Match	0	32768	1	32768
7	Bit Plane 1 + Huffman	Match	4096	32768	1	32768
7	Bit Plane 1 + LZW	Match	225	32768	1	32768
7	Bit Plane 1 + Zero RLC	Match	0	32768	1	32768
7	Bit Plane 2 + Huffman	Match	4097	32768	1	32768
7	Bit Plane 2 + LZW	Match	237	32768	1	32768
7	Bit Plane 2 + Zero RLC	Match	20	32768	1	32768
7	Bit Plane 3 + Huffman	Match	4154	32768	1	32768
7	Bit Plane 3 + LZW	Match	539	32768	1	32768
7	Bit Plane 3 + Zero RLC	Match	728	32768	1	32768
7	Bit Plane 4 + Huffman	Match	4228	32768	1	32768
7	Bit Plane 4 + LZW	Match	804	32768	1	32768
7	Bit Plane 4 + Zero RLC	Match	131072	32768	1	32768
7	Bit Plane 5 + Huffman	Match	7464	32768	1	32768
7	Bit Plane 5 + LZW	Match	3897	32768	1	32768
7	Bit Plane 5 + Zero RLC	Match	67684	32768	1	32768
7	Bit Plane 6 + Huffman	Match	27800	32768	1	32768
7	Bit Plane 6 + LZW	Match	34753	32768	1	32768
7	Bit Plane 6 + Zero RLC	Match	122288	32768	1	32768
7	Bit Plane 7 + Huffman	Match	32318	32768	1	32768
7	Bit Plane 7 + LZW	Match	45870	32768	1	32768
7	Bit Plane 7 + Zero RLC	Match	130624	32768	1	32768
8	Huffman	Match	32780	262144	512	512
8	LZW	Match	830	262144	512	512
8	Predictive + Huffman	Match	32801	262144	512	512
8	Predictive + LZW	Match	885	262144	512	512
8	Bit Plane 0 + Huffman	Match	4096	32768	1	32768
8	Bit Plane 0 + LZW	Match	225	32768	1	32768
8	Bit Plane 0 + Zero RLC	Match	0	32768	1	32768
8	Bit Plane 1 + Huffman	Match	4096	32768	1	32768
8	Bit Plane 1 + LZW	Match	227	32768	1	32768

Image Number	Test Name	Test Result	Compressed Size	Original Size	Rows	Columns
8	Bit Plane 1 + Zero RLC	Match	4	32768	1	32768
8	Bit Plane 2 + Huffman	Match	4097	32768	1	32768
8	Bit Plane 2 + LZW	Match	232	32768	1	32768
8	Bit Plane 2 + Zero RLC	Match	12	32768	1	32768
8	Bit Plane 3 + Huffman	Match	4098	32768	1	32768
8	Bit Plane 3 + LZW	Match	236	32768	1	32768
8	Bit Plane 3 + Zero RLC	Match	20	32768	1	32768
8	Bit Plane 4 + Huffman	Match	4098	32768	1	32768
8	Bit Plane 4 + LZW	Match	240	32768	1	32768
8	Bit Plane 4 + Zero RLC	Match	28	32768	1	32768
8	Bit Plane 5 + Huffman	Match	4102	32768	1	32768
8	Bit Plane 5 + LZW	Match	256	32768	1	32768
8	Bit Plane 5 + Zero RLC	Match	56	32768	1	32768
8	Bit Plane 6 + Huffman	Match	4101	32768	1	32768
8	Bit Plane 6 + LZW	Match	252	32768	1	32768
8	Bit Plane 6 + Zero RLC	Match	48	32768	1	32768
8	Bit Plane 7 + Huffman	Match	4099	32768	1	32768
8	Bit Plane 7 + LZW	Match	242	32768	1	32768
8	Bit Plane 7 + Zero RLC	Match	32	32768	1	32768
9	Huffman	Match	75849	262144	512	512
9	LZW	Match	56752	262144	512	512
9	Predictive + Huffman	Match	97850	262144	512	512
9	Predictive + LZW	Match	68544	262144	512	512
9	Bit Plane 0 + Huffman	Match	4096	32768	1	32768
9	Bit Plane 0 + LZW	Match	225	32768	1	32768
9	Bit Plane 0 + Zero RLC	Match	0	32768	1	32768
9	Bit Plane 1 + Huffman	Match	4097	32768	1	32768
9	Bit Plane 1 + LZW	Match	230	32768	1	32768
9	Bit Plane 1 + Zero RLC	Match	8	32768	1	32768
9	Bit Plane 2 + Huffman	Match	4098	32768	1	32768
9	Bit Plane 2 + LZW	Match	238	32768	1	32768
9	Bit Plane 2 + Zero RLC	Match	20	32768	1	32768
9	Bit Plane 3 + Huffman	Match	4109	32768	1	32768
9	Bit Plane 3 + LZW	Match	302	32768	1	32768
9	Bit Plane 3 + Zero RLC	Match	152	32768	1	32768
9	Bit Plane 4 + Huffman	Match	7775	32768	1	32768
9	Bit Plane 4 + LZW	Match	4328	32768	1	32768
9	Bit Plane 4 + Zero RLC	Match	69656	32768	1	32768
9	Bit Plane 5 + Huffman	Match	12092	32768	1	32768
9	Bit Plane 5 + LZW	Match	12986	32768	1	32768
9	Bit Plane 5 + Zero RLC	Match	131072	32768	1	32768
9	Bit Plane 6 + Huffman	Match	22458	32768	1	32768
9	Bit Plane 6 + LZW	Match	25764	32768	1	32768
9	Bit Plane 6 + Zero RLC	Match	131072	32768	1	32768
9	Bit Plane 7 + Huffman	Match	31904	32768	1	32768
9	Bit Plane 7 + LZW	Match	44983	32768	1	32768

Image Number	Test Name	Test Result	Compressed Size	Original Size	Rows	Columns
9	Bit Plane 7 + Zero RLC	Match	128084	32768	1	32768
10	Huffman	Match	47697	160000	400	400
10	LZW	Match	36722	160000	400	400
10	Predictive + Huffman	Match	61357	160000	400	400
10	Predictive + LZW	Match	42835	160000	400	400
10	Bit Plane 0 + Huffman	Match	2500	20000	1	20000
10	Bit Plane 0 + LZW	Match	169	20000	1	20000
10	Bit Plane 0 + Zero RLC	Match	0	20000	1	20000
10	Bit Plane 1 + Huffman	Match	2500	20000	1	20000
10	Bit Plane 1 + LZW	Match	172	20000	1	20000
10	Bit Plane 1 + Zero RLC	Match	4	20000	1	20000
10	Bit Plane 2 + Huffman	Match	2501	20000	1	20000
10	Bit Plane 2 + LZW	Match	176	20000	1	20000
10	Bit Plane 2 + Zero RLC	Match	12	20000	1	20000
10	Bit Plane 3 + Huffman	Match	3771	20000	1	20000
10	Bit Plane 3 + LZW	Match	598	20000	1	20000
10	Bit Plane 3 + Zero RLC	Match	40072	20000	1	20000
10	Bit Plane 4 + Huffman	Match	3775	20000	1	20000
10	Bit Plane 4 + LZW	Match	615	20000	1	20000
10	Bit Plane 4 + Zero RLC	Match	80000	20000	1	20000
10	Bit Plane 5 + Huffman	Match	14245	20000	1	20000
10	Bit Plane 5 + LZW	Match	16628	20000	1	20000
10	Bit Plane 5 + Zero RLC	Match	79996	20000	1	20000
10	Bit Plane 6 + Huffman	Match	15439	20000	1	20000
10	Bit Plane 6 + LZW	Match	18664	20000	1	20000
10	Bit Plane 6 + Zero RLC	Match	77060	20000	1	20000
10	Bit Plane 7 + Huffman	Match	19831	20000	1	20000
10	Bit Plane 7 + LZW	Match	28543	20000	1	20000
10	Bit Plane 7 + Zero RLC	Match	78860	20000	1	20000
11	Huffman	Match	58182	160000	400	400
11	LZW	Match	65450	160000	400	400
11	Predictive + Huffman	Match	63174	160000	400	400
11	Predictive + LZW	Match	71310	160000	400	400
11	Bit Plane 0 + Huffman	Match	2500	20000	1	20000
11	Bit Plane 0 + LZW	Match	172	20000	1	20000
11	Bit Plane 0 + Zero RLC	Match	4	20000	1	20000
11	Bit Plane 1 + Huffman	Match	2503	20000	1	20000
11	Bit Plane 1 + LZW	Match	188	20000	1	20000
11	Bit Plane 1 + Zero RLC	Match	36	20000	1	20000
11	Bit Plane 2 + Huffman	Match	2544	20000	1	20000
11	Bit Plane 2 + LZW	Match	371	20000	1	20000
11	Bit Plane 2 + Zero RLC	Match	444	20000	1	20000
11	Bit Plane 3 + Huffman	Match	16416	20000	1	20000
11	Bit Plane 3 + LZW	Match	21781	20000	1	20000
11	Bit Plane 3 + Zero RLC	Match	71832	20000	1	20000
11	Bit Plane 4 + Huffman	Match	16330	20000	1	20000

Image Number	Test Name	Test Result	Compressed Size	Original Size	Rows	Columns
11	Bit Plane 4 + LZW	Match	21626	20000	1	20000
11	Bit Plane 4 + Zero RLC	Match	79992	20000	1	20000
11	Bit Plane 5 + Huffman	Match	16030	20000	1	20000
11	Bit Plane 5 + LZW	Match	21200	20000	1	20000
11	Bit Plane 5 + Zero RLC	Match	79996	20000	1	20000
11	Bit Plane 6 + Huffman	Match	19994	20000	1	20000
11	Bit Plane 6 + LZW	Match	29493	20000	1	20000
11	Bit Plane 6 + Zero RLC	Match	79768	20000	1	20000
11	Bit Plane 7 + Huffman	Match	19998	20000	1	20000
11	Bit Plane 7 + LZW	Match	29490	20000	1	20000
11	Bit Plane 7 + Zero RLC	Match	79752	20000	1	20000
12	Huffman	Match	32776	262144	512	512
12	LZW	Match	819	262144	512	512
12	Predictive + Huffman	Match	32789	262144	512	512
12	Predictive + LZW	Match	855	262144	512	512
12	Bit Plane 0 + Huffman	Match	4096	32768	1	32768
12	Bit Plane 0 + LZW	Match	225	32768	1	32768
12	Bit Plane 0 + Zero RLC	Match	0	32768	1	32768
12	Bit Plane 1 + Huffman	Match	4096	32768	1	32768
12	Bit Plane 1 + LZW	Match	225	32768	1	32768
12	Bit Plane 1 + Zero RLC	Match	0	32768	1	32768
12	Bit Plane 2 + Huffman	Match	4096	32768	1	32768
12	Bit Plane 2 + LZW	Match	228	32768	1	32768
12	Bit Plane 2 + Zero RLC	Match	4	32768	1	32768
12	Bit Plane 3 + Huffman	Match	4096	32768	1	32768
12	Bit Plane 3 + LZW	Match	230	32768	1	32768
12	Bit Plane 3 + Zero RLC	Match	8	32768	1	32768
12	Bit Plane 4 + Huffman	Match	4098	32768	1	32768
12	Bit Plane 4 + LZW	Match	239	32768	1	32768
12	Bit Plane 4 + Zero RLC	Match	20	32768	1	32768
12	Bit Plane 5 + Huffman	Match	4097	32768	1	32768
12	Bit Plane 5 + LZW	Match	233	32768	1	32768
12	Bit Plane 5 + Zero RLC	Match	16	32768	1	32768
12	Bit Plane 6 + Huffman	Match	4098	32768	1	32768
12	Bit Plane 6 + LZW	Match	240	32768	1	32768
12	Bit Plane 6 + Zero RLC	Match	24	32768	1	32768
12	Bit Plane 7 + Huffman	Match	4099	32768	1	32768
12	Bit Plane 7 + LZW	Match	244	32768	1	32768
12	Bit Plane 7 + Zero RLC	Match	28	32768	1	32768
13	Huffman	Match	67454	262144	512	512
13	LZW	Match	51698	262144	512	512
13	Predictive + Huffman	Match	111344	262144	512	512
13	Predictive + LZW	Match	90891	262144	512	512
13	Bit Plane 0 + Huffman	Match	4096	32768	1	32768
13	Bit Plane 0 + LZW	Match	225	32768	1	32768
13	Bit Plane 0 + Zero RLC	Match	0	32768	1	32768

Image Number	Test Name	Test Result	Compressed Size	Original Size	Rows	Columns
13	Bit Plane 1 + Huffman	Match	4096	32768	1	32768
13	Bit Plane 1 + LZW	Match	227	32768	1	32768
13	Bit Plane 1 + Zero RLC	Match	4	32768	1	32768
13	Bit Plane 2 + Huffman	Match	4098	32768	1	32768
13	Bit Plane 2 + LZW	Match	243	32768	1	32768
13	Bit Plane 2 + Zero RLC	Match	28	32768	1	32768
13	Bit Plane 3 + Huffman	Match	8250	32768	1	32768
13	Bit Plane 3 + LZW	Match	5273	32768	1	32768
13	Bit Plane 3 + Zero RLC	Match	66420	32768	1	32768
13	Bit Plane 4 + Huffman	Match	8281	32768	1	32768
13	Bit Plane 4 + LZW	Match	5322	32768	1	32768
13	Bit Plane 4 + Zero RLC	Match	131072	32768	1	32768
13	Bit Plane 5 + Huffman	Match	14014	32768	1	32768
13	Bit Plane 5 + LZW	Match	17118	32768	1	32768
13	Bit Plane 5 + Zero RLC	Match	66312	32768	1	32768
13	Bit Plane 6 + Huffman	Match	29026	32768	1	32768
13	Bit Plane 6 + LZW	Match	38681	32768	1	32768
13	Bit Plane 6 + Zero RLC	Match	131032	32768	1	32768
13	Bit Plane 7 + Huffman	Match	12031	32768	1	32768
13	Bit Plane 7 + LZW	Match	10506	32768	1	32768
13	Bit Plane 7 + Zero RLC	Match	131072	32768	1	32768
14	Huffman	Match	492465	1310720	1024	1280
14	LZW	Match	483940	1310720	1024	1280
14	Predictive + Huffman	Match	692172	1310720	1024	1280
14	Predictive + LZW	Match	703088	1310720	1024	1280
14	Bit Plane 0 + Huffman	Match	20482	163840	1	163840
14	Bit Plane 0 + LZW	Match	608	163840	1	163840
14	Bit Plane 0 + Zero RLC	Match	28	163840	1	163840
14	Bit Plane 1 + Huffman	Match	20510	163840	1	163840
14	Bit Plane 1 + LZW	Match	758	163840	1	163840
14	Bit Plane 1 + Zero RLC	Match	312	163840	1	163840
14	Bit Plane 2 + Huffman	Match	21906	163840	1	163840
14	Bit Plane 2 + LZW	Match	6568	163840	1	163840
14	Bit Plane 2 + Zero RLC	Match	17008	163840	1	163840
14	Bit Plane 3 + Huffman	Match	124119	163840	1	163840
14	Bit Plane 3 + LZW	Match	138348	163840	1	163840
14	Bit Plane 3 + Zero RLC	Match	609432	163840	1	163840
14	Bit Plane 4 + Huffman	Match	143286	163840	1	163840
14	Bit Plane 4 + LZW	Match	174735	163840	1	163840
14	Bit Plane 4 + Zero RLC	Match	654008	163840	1	163840
14	Bit Plane 5 + Huffman	Match	150258	163840	1	163840
14	Bit Plane 5 + LZW	Match	191981	163840	1	163840
14	Bit Plane 5 + Zero RLC	Match	655152	163840	1	163840
14	Bit Plane 6 + Huffman	Match	163840	163840	1	163840
14	Bit Plane 6 + LZW	Match	213869	163840	1	163840
14	Bit Plane 6 + Zero RLC	Match	651924	163840	1	163840

Image Number	Test Name	Test Result	Compressed Size	Original Size	Rows	Columns
14	Bit Plane 7 + Huffman	Match	140139	163840	1	163840
14	Bit Plane 7 + LZW	Match	175298	163840	1	163840
14	Bit Plane 7 + Zero RLC	Match	654816	163840	1	163840
15	Huffman	Match	528083	1310720	1024	1280
15	LZW	Match	551236	1310720	1024	1280
15	Predictive + Huffman	Match	685579	1310720	1024	1280
15	Predictive + LZW	Match	771372	1310720	1024	1280
15	Bit Plane 0 + Huffman	Match	20485	163840	1	163840
15	Bit Plane 0 + LZW	Match	621	163840	1	163840
15	Bit Plane 0 + Zero RLC	Match	60	163840	1	163840
15	Bit Plane 1 + Huffman	Match	20530	163840	1	163840
15	Bit Plane 1 + LZW	Match	846	163840	1	163840
15	Bit Plane 1 + Zero RLC	Match	488	163840	1	163840
15	Bit Plane 2 + Huffman	Match	21021	163840	1	163840
15	Bit Plane 2 + LZW	Match	3097	163840	1	163840
15	Bit Plane 2 + Zero RLC	Match	5500	163840	1	163840
15	Bit Plane 3 + Huffman	Match	42864	163840	1	163840
15	Bit Plane 3 + LZW	Match	47001	163840	1	163840
15	Bit Plane 3 + Zero RLC	Match	170636	163840	1	163840
15	Bit Plane 4 + Huffman	Match	156593	163840	1	163840
15	Bit Plane 4 + LZW	Match	200379	163840	1	163840
15	Bit Plane 4 + Zero RLC	Match	625512	163840	1	163840
15	Bit Plane 5 + Huffman	Match	163629	163840	1	163840
15	Bit Plane 5 + LZW	Match	212881	163840	1	163840
15	Bit Plane 5 + Zero RLC	Match	653460	163840	1	163840
15	Bit Plane 6 + Huffman	Match	163840	163840	1	163840
15	Bit Plane 6 + LZW	Match	214030	163840	1	163840
15	Bit Plane 6 + Zero RLC	Match	652724	163840	1	163840
15	Bit Plane 7 + Huffman	Match	151708	163840	1	163840
15	Bit Plane 7 + LZW	Match	188711	163840	1	163840
15	Bit Plane 7 + Zero RLC	Match	644616	163840	1	163840
16	Huffman	Match	244522	1310720	1024	1280
16	LZW	Match	225263	1310720	1024	1280
16	Predictive + Huffman	Match	271719	1310720	1024	1280
16	Predictive + LZW	Match	269890	1310720	1024	1280
16	Bit Plane 0 + Huffman	Match	20480	163840	1	163840
16	Bit Plane 0 + LZW	Match	588	163840	1	163840
16	Bit Plane 0 + Zero RLC	Match	0	163840	1	163840
16	Bit Plane 1 + Huffman	Match	20480	163840	1	163840
16	Bit Plane 1 + LZW	Match	588	163840	1	163840
16	Bit Plane 1 + Zero RLC	Match	0	163840	1	163840
16	Bit Plane 2 + Huffman	Match	20480	163840	1	163840
16	Bit Plane 2 + LZW	Match	594	163840	1	163840
16	Bit Plane 2 + Zero RLC	Match	8	163840	1	163840
16	Bit Plane 3 + Huffman	Match	20483	163840	1	163840
16	Bit Plane 3 + LZW	Match	612	163840	1	163840

Image Number	Test Name	Test Result	Compressed Size	Original Size	Rows	Columns
16	Bit Plane 3 + Zero RLC	Match	32	163840	1	163840
16	Bit Plane 4 + Huffman	Match	20505	163840	1	163840
16	Bit Plane 4 + LZW	Match	720	163840	1	163840
16	Bit Plane 4 + Zero RLC	Match	232	163840	1	163840
16	Bit Plane 5 + Huffman	Match	125625	163840	1	163840
16	Bit Plane 5 + LZW	Match	143929	163840	1	163840
16	Bit Plane 5 + Zero RLC	Match	548504	163840	1	163840
16	Bit Plane 6 + Huffman	Match	125920	163840	1	163840
16	Bit Plane 6 + LZW	Match	144692	163840	1	163840
16	Bit Plane 6 + Zero RLC	Match	655280	163840	1	163840
16	Bit Plane 7 + Huffman	Match	153132	163840	1	163840
16	Bit Plane 7 + LZW	Match	193660	163840	1	163840
16	Bit Plane 7 + Zero RLC	Match	655116	163840	1	163840
17	Huffman	Match	462875	1310720	1024	1280
17	LZW	Match	497673	1310720	1024	1280
17	Predictive + Huffman	Match	504069	1310720	1024	1280
17	Predictive + LZW	Match	545074	1310720	1024	1280
17	Bit Plane 0 + Huffman	Match	20482	163840	1	163840
17	Bit Plane 0 + LZW	Match	606	163840	1	163840
17	Bit Plane 0 + Zero RLC	Match	24	163840	1	163840
17	Bit Plane 1 + Huffman	Match	20490	163840	1	163840
17	Bit Plane 1 + LZW	Match	651	163840	1	163840
17	Bit Plane 1 + Zero RLC	Match	100	163840	1	163840
17	Bit Plane 2 + Huffman	Match	20518	163840	1	163840
17	Bit Plane 2 + LZW	Match	779	163840	1	163840
17	Bit Plane 2 + Zero RLC	Match	332	163840	1	163840
17	Bit Plane 3 + Huffman	Match	20702	163840	1	163840
17	Bit Plane 3 + LZW	Match	1651	163840	1	163840
17	Bit Plane 3 + Zero RLC	Match	2052	163840	1	163840
17	Bit Plane 4 + Huffman	Match	25733	163840	1	163840
17	Bit Plane 4 + LZW	Match	16466	163840	1	163840
17	Bit Plane 4 + Zero RLC	Match	49752	163840	1	163840
17	Bit Plane 5 + Huffman	Match	126508	163840	1	163840
17	Bit Plane 5 + LZW	Match	155161	163840	1	163840
17	Bit Plane 5 + Zero RLC	Match	558896	163840	1	163840
17	Bit Plane 6 + Huffman	Match	161718	163840	1	163840
17	Bit Plane 6 + LZW	Match	209024	163840	1	163840
17	Bit Plane 6 + Zero RLC	Match	646680	163840	1	163840
17	Bit Plane 7 + Huffman	Match	163106	163840	1	163840
17	Bit Plane 7 + LZW	Match	211523	163840	1	163840
17	Bit Plane 7 + Zero RLC	Match	649180	163840	1	163840
18	Huffman	Match	260236	1310720	1024	1280
18	LZW	Match	253840	1310720	1024	1280
18	Predictive + Huffman	Match	289700	1310720	1024	1280
18	Predictive + LZW	Match	297732	1310720	1024	1280
18	Bit Plane 0 + Huffman	Match	20480	163840	1	163840

Image Number	Test Name	Test Result	Compressed Size	Original Size	Rows	Columns
18	Bit Plane 0 + LZW	Match	588	163840	1	163840
18	Bit Plane 0 + Zero RLC	Match	0	163840	1	163840
18	Bit Plane 1 + Huffman	Match	20480	163840	1	163840
18	Bit Plane 1 + LZW	Match	588	163840	1	163840
18	Bit Plane 1 + Zero RLC	Match	0	163840	1	163840
18	Bit Plane 2 + Huffman	Match	20480	163840	1	163840
18	Bit Plane 2 + LZW	Match	591	163840	1	163840
18	Bit Plane 2 + Zero RLC	Match	4	163840	1	163840
18	Bit Plane 3 + Huffman	Match	20483	163840	1	163840
18	Bit Plane 3 + LZW	Match	612	163840	1	163840
18	Bit Plane 3 + Zero RLC	Match	32	163840	1	163840
18	Bit Plane 4 + Huffman	Match	20512	163840	1	163840
18	Bit Plane 4 + LZW	Match	755	163840	1	163840
18	Bit Plane 4 + Zero RLC	Match	308	163840	1	163840
18	Bit Plane 5 + Huffman	Match	127135	163840	1	163840
18	Bit Plane 5 + LZW	Match	145588	163840	1	163840
18	Bit Plane 5 + Zero RLC	Match	561468	163840	1	163840
18	Bit Plane 6 + Huffman	Match	127195	163840	1	163840
18	Bit Plane 6 + LZW	Match	146687	163840	1	163840
18	Bit Plane 6 + Zero RLC	Match	655292	163840	1	163840
18	Bit Plane 7 + Huffman	Match	153324	163840	1	163840
18	Bit Plane 7 + LZW	Match	195116	163840	1	163840
18	Bit Plane 7 + Zero RLC	Match	655140	163840	1	163840
19	Huffman	Match	203141	262144	512	512
19	LZW	Match	197166	262144	512	512
19	Predictive + Huffman	Match	145977	262144	512	512
19	Predictive + LZW	Match	169122	262144	512	512
19	Bit Plane 0 + Huffman	Match	5605	32768	1	32768
19	Bit Plane 0 + LZW	Match	2556	32768	1	32768
19	Bit Plane 0 + Zero RLC	Match	10948	32768	1	32768
19	Bit Plane 1 + Huffman	Match	9758	32768	1	32768
19	Bit Plane 1 + LZW	Match	8826	32768	1	32768
19	Bit Plane 1 + Zero RLC	Match	126060	32768	1	32768
19	Bit Plane 2 + Huffman	Match	18172	32768	1	32768
19	Bit Plane 2 + LZW	Match	19949	32768	1	32768
19	Bit Plane 2 + Zero RLC	Match	72596	32768	1	32768
19	Bit Plane 3 + Huffman	Match	26107	32768	1	32768
19	Bit Plane 3 + LZW	Match	32184	32768	1	32768
19	Bit Plane 3 + Zero RLC	Match	110020	32768	1	32768
19	Bit Plane 4 + Huffman	Match	32424	32768	1	32768
19	Bit Plane 4 + LZW	Match	46119	32768	1	32768
19	Bit Plane 4 + Zero RLC	Match	128740	32768	1	32768
19	Bit Plane 5 + Huffman	Match	32768	32768	1	32768
19	Bit Plane 5 + LZW	Match	48041	32768	1	32768
19	Bit Plane 5 + Zero RLC	Match	130364	32768	1	32768
19	Bit Plane 6 + Huffman	Match	32768	32768	1	32768

Image Number	Test Name	Test Result	Compressed Size	Original Size	Rows	Columns
19	Bit Plane 6 + LZW	Match	47998	32768	1	32768
19	Bit Plane 6 + Zero RLC	Match	130412	32768	1	32768
19	Bit Plane 7 + Huffman	Match	32768	32768	1	32768
19	Bit Plane 7 + LZW	Match	48000	32768	1	32768
19	Bit Plane 7 + Zero RLC	Match	130572	32768	1	32768
20	Huffman	Match	128123	262144	512	512
20	LZW	Match	133076	262144	512	512
20	Predictive + Huffman	Match	114614	262144	512	512
20	Predictive + LZW	Match	129213	262144	512	512
20	Bit Plane 0 + Huffman	Match	4096	32768	1	32768
20	Bit Plane 0 + LZW	Match	225	32768	1	32768
20	Bit Plane 0 + Zero RLC	Match	0	32768	1	32768
20	Bit Plane 1 + Huffman	Match	4096	32768	1	32768
20	Bit Plane 1 + LZW	Match	228	32768	1	32768
20	Bit Plane 1 + Zero RLC	Match	4	32768	1	32768
20	Bit Plane 2 + Huffman	Match	4844	32768	1	32768
20	Bit Plane 2 + LZW	Match	1428	32768	1	32768
20	Bit Plane 2 + Zero RLC	Match	4276	32768	1	32768
20	Bit Plane 3 + Huffman	Match	23602	32768	1	32768
20	Bit Plane 3 + LZW	Match	29651	32768	1	32768
20	Bit Plane 3 + Zero RLC	Match	129456	32768	1	32768
20	Bit Plane 4 + Huffman	Match	26624	32768	1	32768
20	Bit Plane 4 + LZW	Match	34710	32768	1	32768
20	Bit Plane 4 + Zero RLC	Match	107640	32768	1	32768
20	Bit Plane 5 + Huffman	Match	32545	32768	1	32768
20	Bit Plane 5 + LZW	Match	46876	32768	1	32768
20	Bit Plane 5 + Zero RLC	Match	129168	32768	1	32768
20	Bit Plane 6 + Huffman	Match	32768	32768	1	32768
20	Bit Plane 6 + LZW	Match	48039	32768	1	32768
20	Bit Plane 6 + Zero RLC	Match	130528	32768	1	32768
20	Bit Plane 7 + Huffman	Match	32768	32768	1	32768
20	Bit Plane 7 + LZW	Match	48011	32768	1	32768
20	Bit Plane 7 + Zero RLC	Match	130572	32768	1	32768
21	Huffman	Match	180277	262144	512	512
21	LZW	Match	105979	262144	512	512
21	Predictive + Huffman	Match	72801	262144	512	512
21	Predictive + LZW	Match	76458	262144	512	512
21	Bit Plane 0 + Huffman	Match	4491	32768	1	32768
21	Bit Plane 0 + LZW	Match	784	32768	1	32768
21	Bit Plane 0 + Zero RLC	Match	8308	32768	1	32768
21	Bit Plane 1 + Huffman	Match	5084	32768	1	32768
21	Bit Plane 1 + LZW	Match	1641	32768	1	32768
21	Bit Plane 1 + Zero RLC	Match	15020	32768	1	32768
21	Bit Plane 2 + Huffman	Match	7886	32768	1	32768
21	Bit Plane 2 + LZW	Match	5162	32768	1	32768
21	Bit Plane 2 + Zero RLC	Match	42536	32768	1	32768

Image Number	Test Name	Test Result	Compressed Size	Original Size	Rows	Columns
21	Bit Plane 3 + Huffman	Match	9035	32768	1	32768
21	Bit Plane 3 + LZW	Match	7168	32768	1	32768
21	Bit Plane 3 + Zero RLC	Match	112232	32768	1	32768
21	Bit Plane 4 + Huffman	Match	15256	32768	1	32768
21	Bit Plane 4 + LZW	Match	15306	32768	1	32768
21	Bit Plane 4 + Zero RLC	Match	70360	32768	1	32768
21	Bit Plane 5 + Huffman	Match	23655	32768	1	32768
21	Bit Plane 5 + LZW	Match	28582	32768	1	32768
21	Bit Plane 5 + Zero RLC	Match	105776	32768	1	32768
21	Bit Plane 6 + Huffman	Match	30812	32768	1	32768
21	Bit Plane 6 + LZW	Match	42290	32768	1	32768
21	Bit Plane 6 + Zero RLC	Match	126516	32768	1	32768
21	Bit Plane 7 + Huffman	Match	32494	32768	1	32768
21	Bit Plane 7 + LZW	Match	46635	32768	1	32768
21	Bit Plane 7 + Zero RLC	Match	130452	32768	1	32768
22	Huffman	Match	178867	262144	512	512
22	LZW	Match	143245	262144	512	512
22	Predictive + Huffman	Match	101638	262144	512	512
22	Predictive + LZW	Match	113702	262144	512	512
22	Bit Plane 0 + Huffman	Match	4421	32768	1	32768
22	Bit Plane 0 + LZW	Match	783	32768	1	32768
22	Bit Plane 0 + Zero RLC	Match	5104	32768	1	32768
22	Bit Plane 1 + Huffman	Match	5862	32768	1	32768
22	Bit Plane 1 + LZW	Match	2641	32768	1	32768
22	Bit Plane 1 + Zero RLC	Match	21220	32768	1	32768
22	Bit Plane 2 + Huffman	Match	6208	32768	1	32768
22	Bit Plane 2 + LZW	Match	3421	32768	1	32768
22	Bit Plane 2 + Zero RLC	Match	119332	32768	1	32768
22	Bit Plane 3 + Huffman	Match	11076	32768	1	32768
22	Bit Plane 3 + LZW	Match	10301	32768	1	32768
22	Bit Plane 3 + Zero RLC	Match	44876	32768	1	32768
22	Bit Plane 4 + Huffman	Match	24543	32768	1	32768
22	Bit Plane 4 + LZW	Match	29870	32768	1	32768
22	Bit Plane 4 + Zero RLC	Match	110312	32768	1	32768
22	Bit Plane 5 + Huffman	Match	30935	32768	1	32768
22	Bit Plane 5 + LZW	Match	42052	32768	1	32768
22	Bit Plane 5 + Zero RLC	Match	126860	32768	1	32768
22	Bit Plane 6 + Huffman	Match	32768	32768	1	32768
22	Bit Plane 6 + LZW	Match	47913	32768	1	32768
22	Bit Plane 6 + Zero RLC	Match	130424	32768	1	32768
22	Bit Plane 7 + Huffman	Match	32768	32768	1	32768
22	Bit Plane 7 + LZW	Match	48063	32768	1	32768
22	Bit Plane 7 + Zero RLC	Match	130540	32768	1	32768

Figure G-7: Compression Program Raw Results